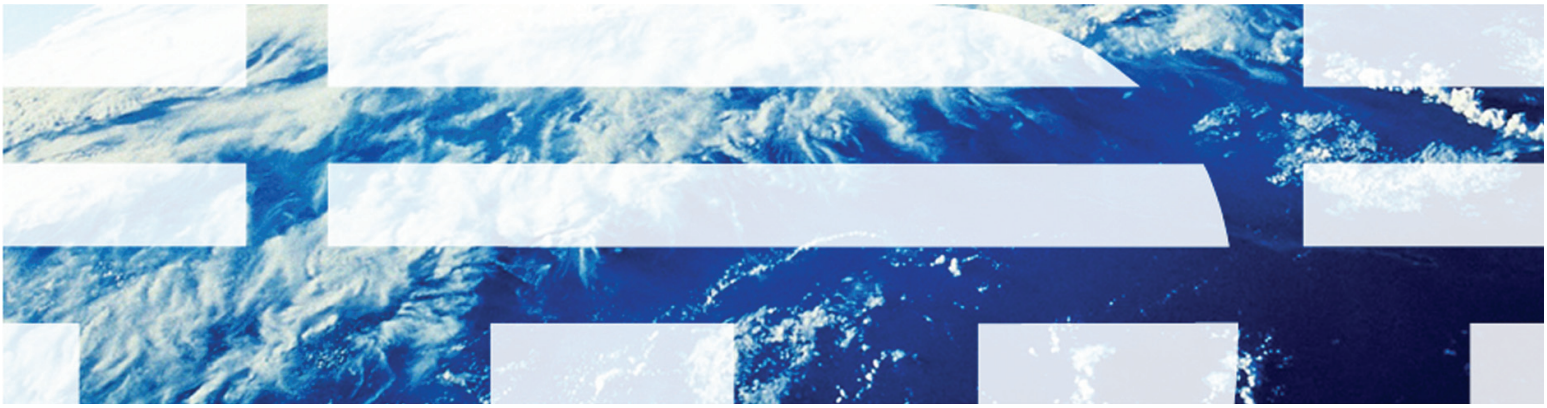


IBM Tivoli NetView for z/OS 6.1: NetView PIPEs

Student's Training Guide

Course: TZ233 ERC: 1.0

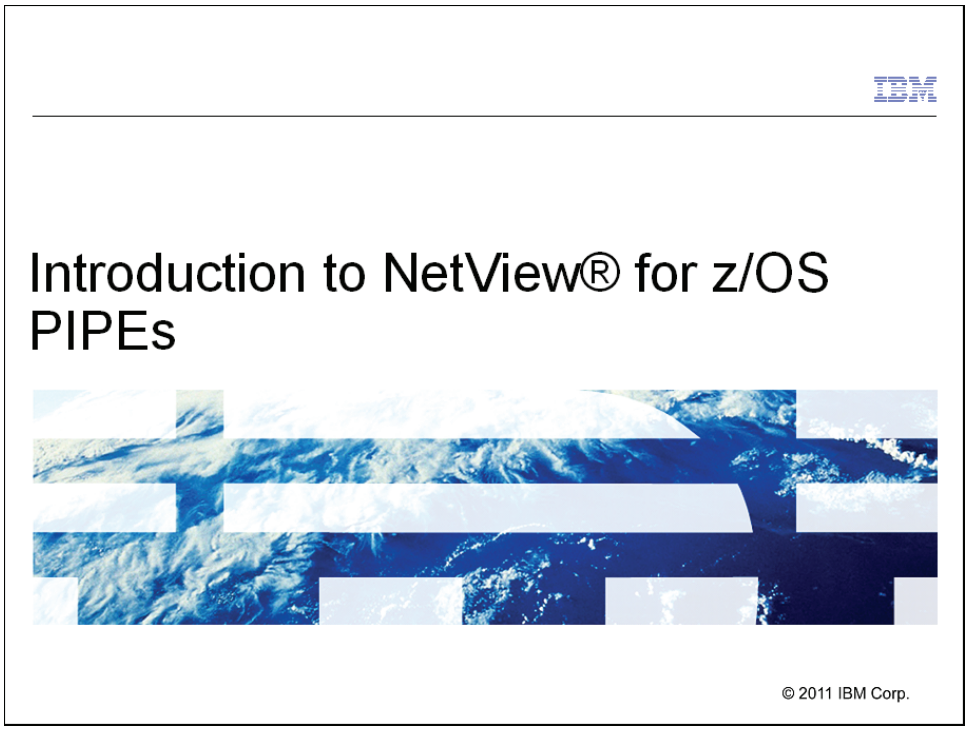
September 2011

Printed in Ireland

Table of contents

Unit 1: Introduction to NetView® for z/OS PIPEs

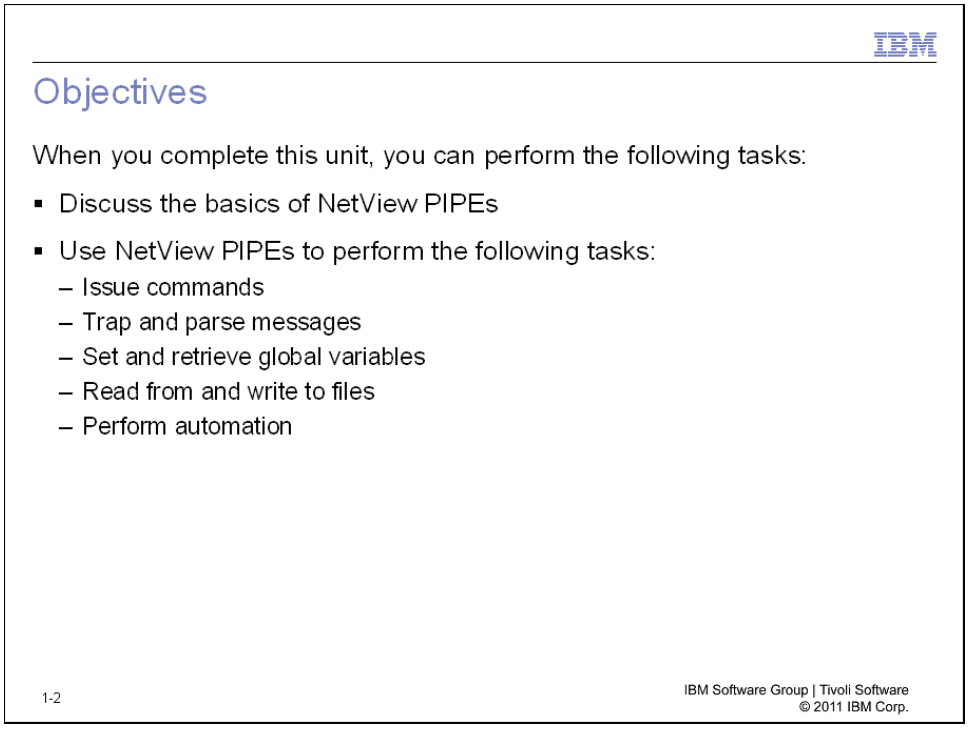
Unit 1: Introduction to NetView® for z/OS PIPEs



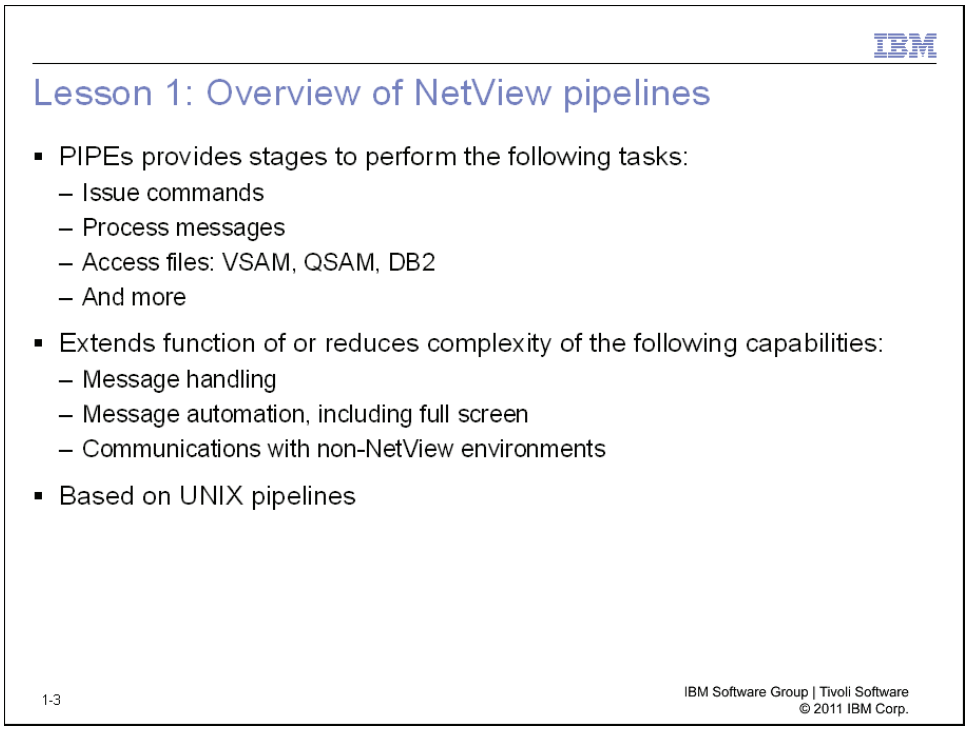
Introduction

This unit provides basic information of NetView® for z/OS® PIPEs and the most commonly used PIPE stages. You learn to use PIPE to perform tasks as listed in the slide.

Objectives



Lesson 1: Overview of NetView pipelines

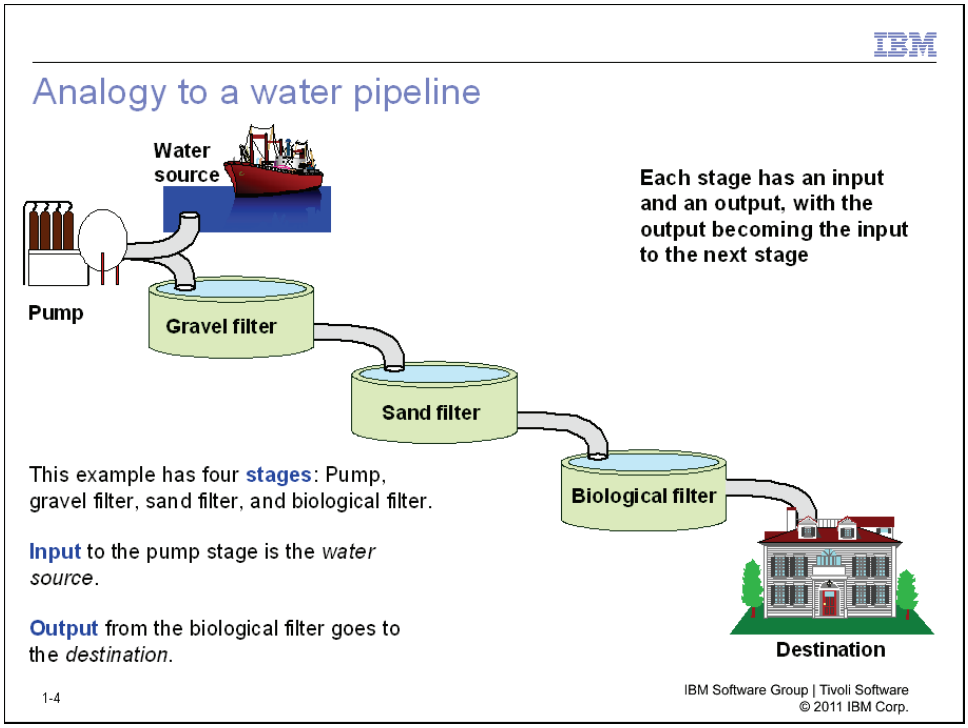


NetView pipelines help you solve a complex problem by dividing the problem into a set of smaller, simpler steps. Each step or stage handles one part of the overall problem. PIPE stages can do the following tasks:

- Read data from several sources. For example, it can read files on disk or variables in command procedures.
- Filter and refine the data.
- Export (output) the data from the pipeline.

You can connect stages in logical sequence until they collectively cover all steps that are required to solve your problem.

Analogy to a water pipeline

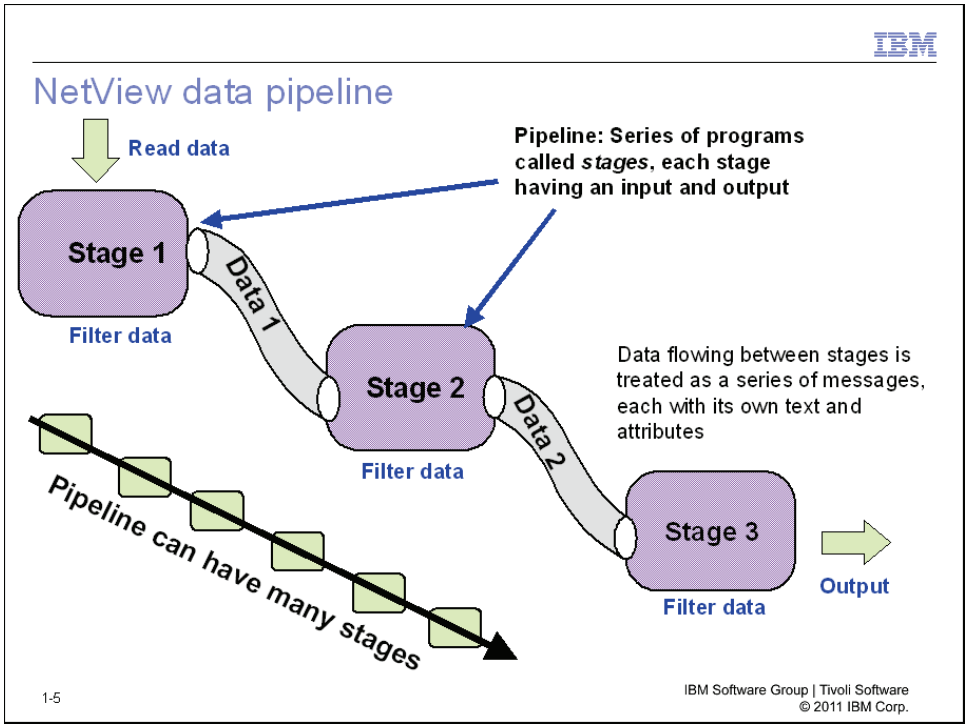


You can understand NetView pipelines if you compare them to a plumbing pipeline in a water treatment process:

Table 1: Comparing plumbing pipeline to NetView pipeline

Plumbing pipeline	NetView pipeline
Receives water from a source. An example is a reservoir or a well.	Receives data from a source. An example is a keyboard or disk.
Passes water through system.	Passes data through stages.
Combines different sizes and shapes of filters to perform complex purification processes.	Combines different stage specifications to perform complex data refinement.
Delivers purified water.	Delivers refined data to other programs or storage.

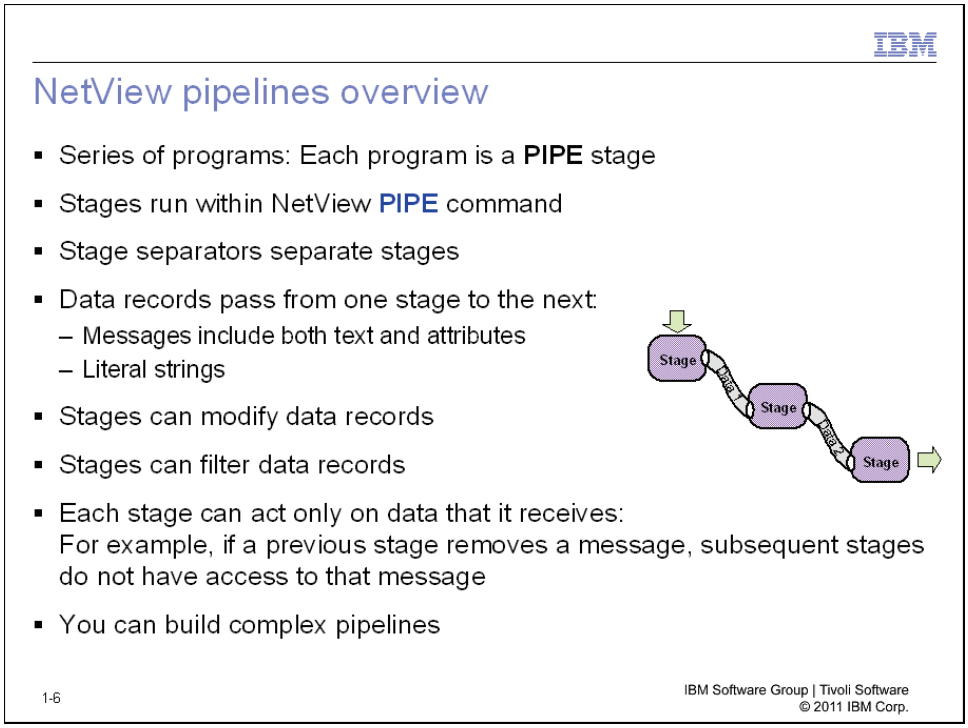
NetView data pipeline



Input data passes through a stage, and a stage command performs an action on it. Several of these stages can be linked together to form a pipeline that produces the output stream that is required.

Each stage in the pipeline acts on the data in the pipeline. For example, stages can edit, delete, add, or display the pipeline data. Each stage has an *input stream* and an *output stream*. When a stage ends, its output stream becomes the input stream for the next stage.

NetView pipelines overview

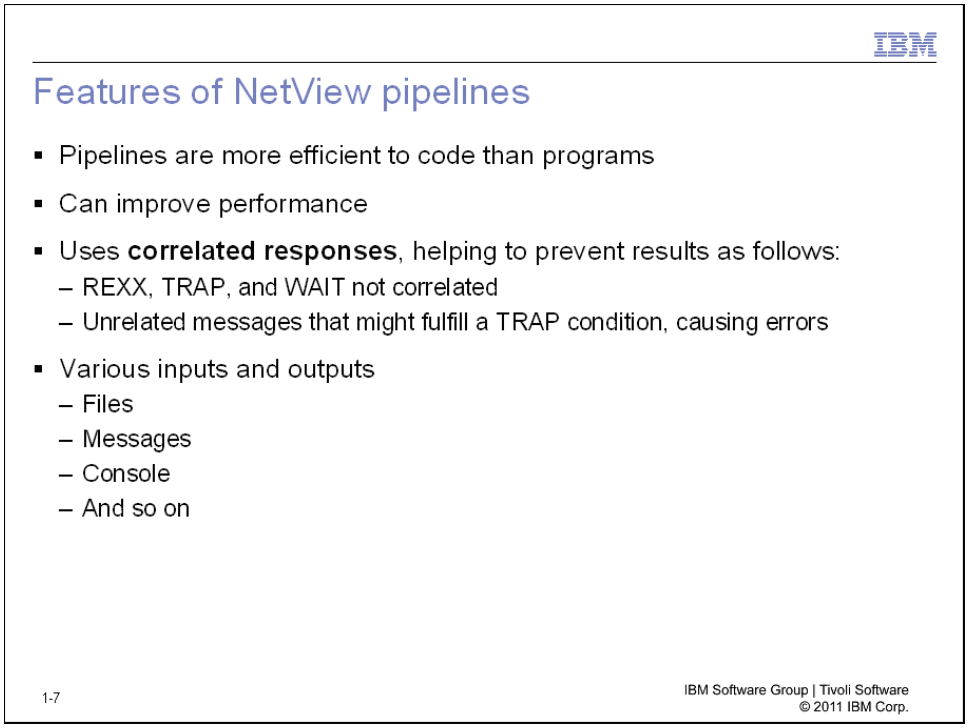


Data in the pipeline is a series of discrete records called messages. After it is read into the pipeline, each record becomes a message consisting of message text and message attributes. Data can be discarded, modified, split, ignored, logged, displayed, colored, and so on.

The processing is performed by *pipeline stages*. The *output stream* of one stage is the *input stream* to the next stage. Some stages support two or more input and output streams.

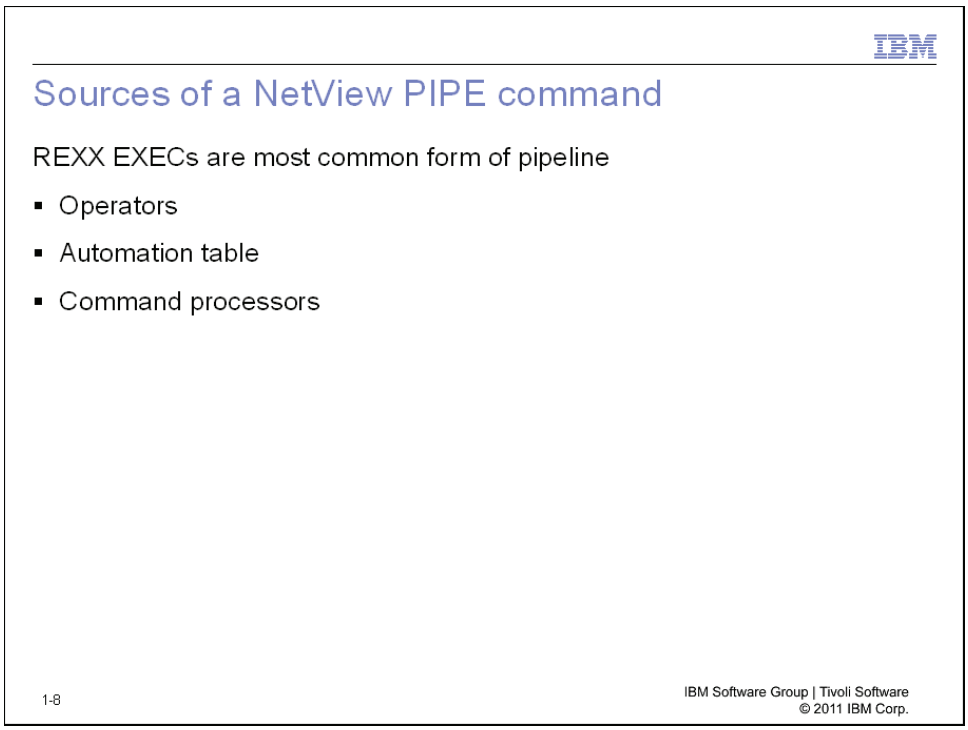
A multiline message is one data record. Multiline messages can be split into multiple single-line data records. Discussions about more complex pipelines (for example, nested and multi-stage pipelines) occur later in this unit.

Features of NetView pipelines



Pipelines are typically more efficient to code and test than programs. Pipeline data results in correlated asynchronous responses.

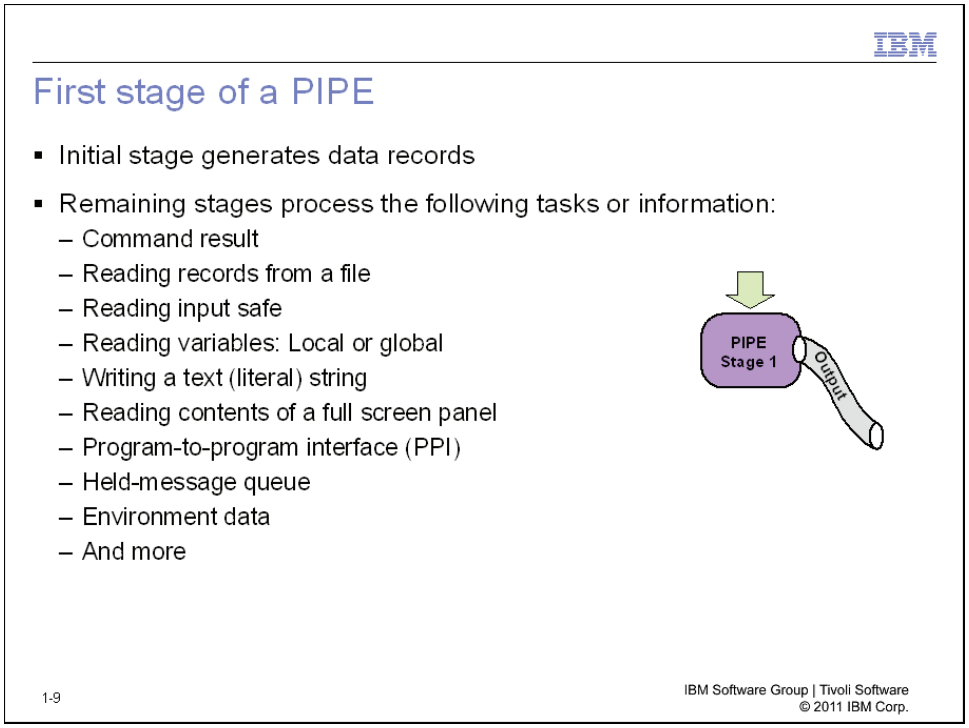
Sources of a NetView PIPE command



Many pipelines start out simple, but can quickly become complicated. Note that there is a difference between a *simple* pipeline and a *complex* pipeline. A complex pipeline has a slightly different structure than the simple type. Discussion of complex pipes occurs later in this unit.

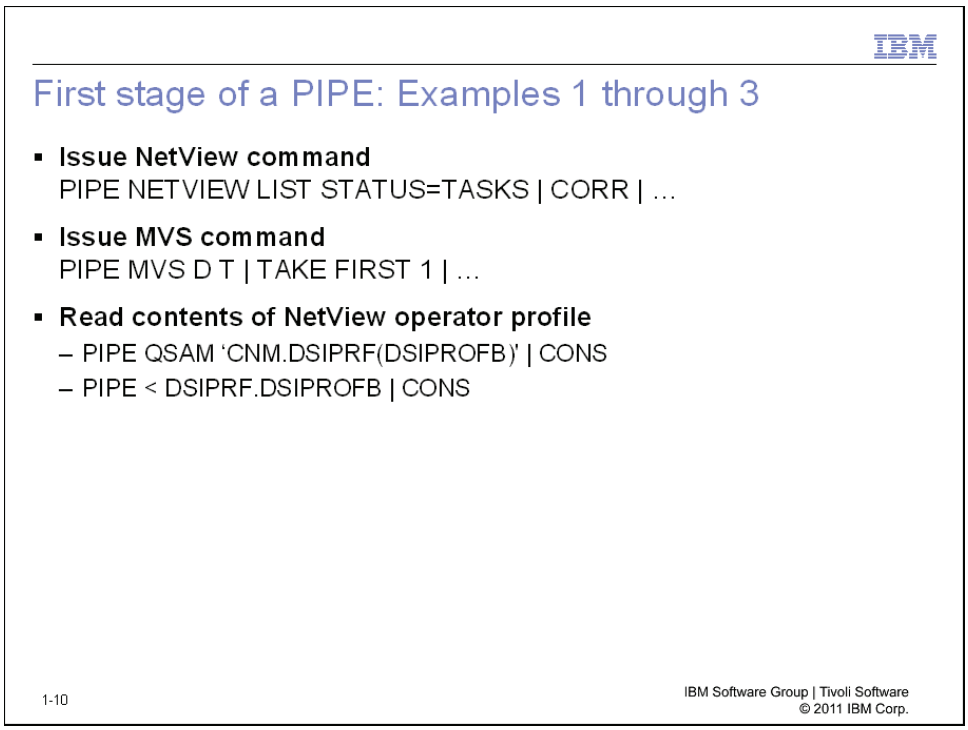
Operators can issue pipelines. REXX EXECs are the most common form of pipelines because the pipeline stages often require programming logic.

First stage of a PIPE



The first stage of a pipeline typically issues a command to generate data records for the remaining stages to process. This slide shows several examples of creating data for a pipeline.

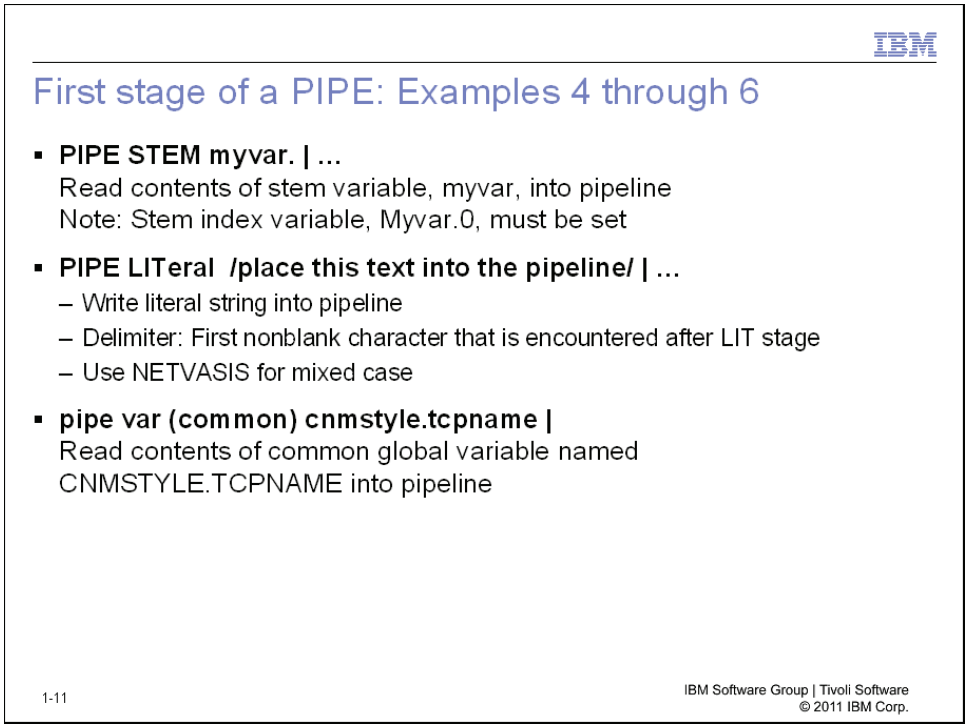
First stage of a PIPE: Examples 1 through 3



Using NetView pipelines, you can perform tasks as follows:

- Issue commands. Examples are NetView and MVS commands.
- Read data from files and write data to files. Pipeline stages are provided to support VSAM, QSAM, and DB2® or SQL files. Discussion of the QSAM and read from (<) pipe stages occur later in this unit.

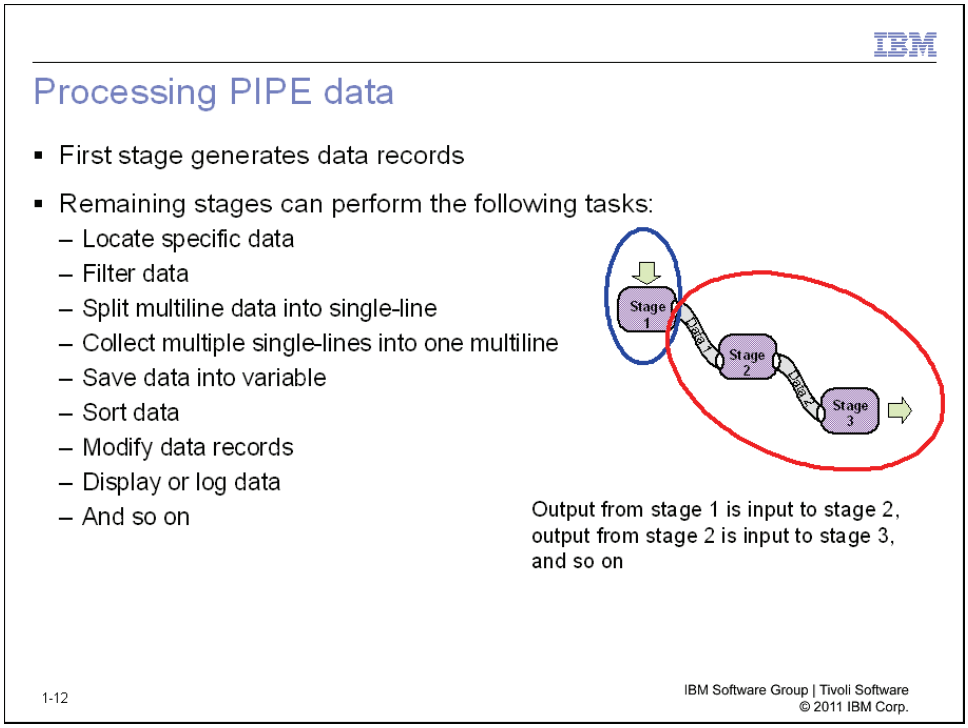
First stage of a PIPE: Examples 4 through 6



This slide shows examples of generating data records as the first stage of a pipeline. If LITERAL is not the first stage, messages remain in the pipeline, and text added by this stage is inserted in front of the existing messages. You can change the order of pipeline data with the REVERSE or APPEND stages.

Note: You cannot issue STEM and VAR stages from the NetView command line.

Processing PIPE data



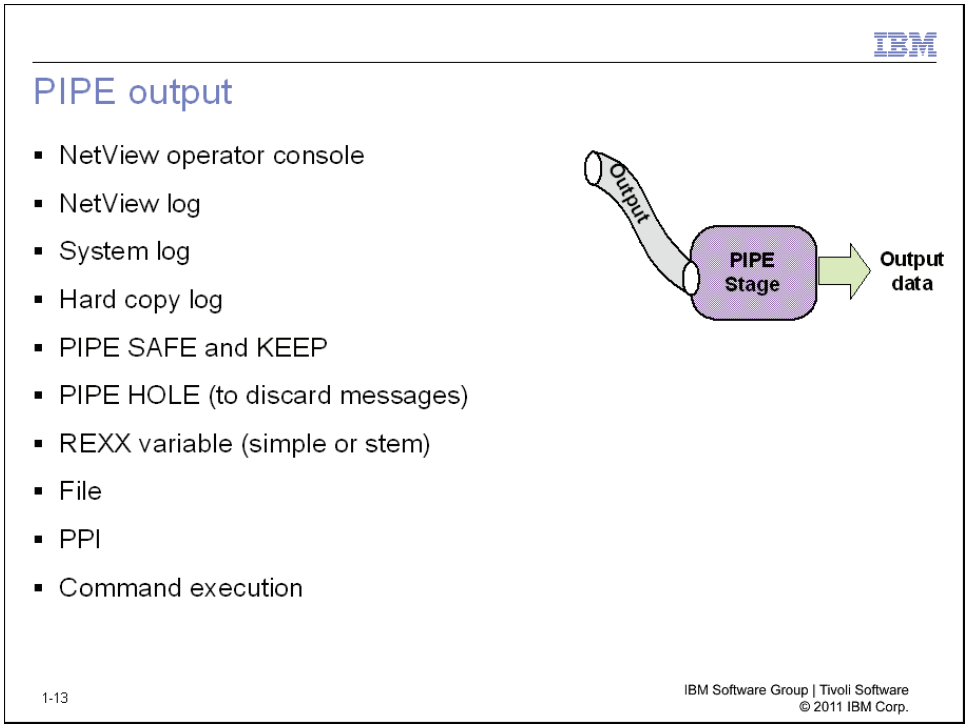
The first stage in a pipeline generates data for its output stream to pass to the second stage in the pipeline. The remaining pipeline stages can locate data, sort data, modify data, and so on, passing the data in the output stream for each stage.

Pipeline stages are categorized as follows:

- Device drivers: Transport data but do not take any action on it. Device drivers write their input stream to their output stream. PIPE QSAM and PIPE NETVIEW are examples of device drivers.
- Filters: Examine their input stream, passing data that matches a specified criteria to their *primary output stream*. Data that does not match the specified criteria passes to a *secondary output stream*. PIPE LOCATE and PIPE TAKE are examples of filters.

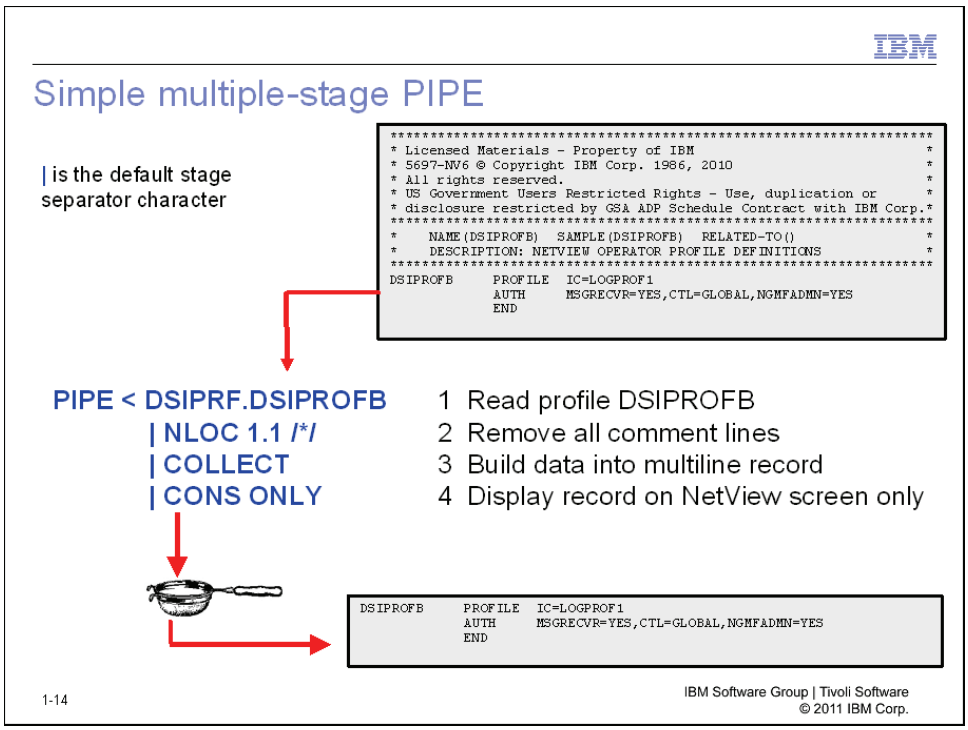
Pipelines can handle a large amount of data without the need for all of the data to be in storage simultaneously. Some stages (for example, COLLECT) might need to handle all of the data simultaneously.

PIPE output



Pipeline data can be displayed, logged, discarded, written to a file, and so on. Note that the slide shows the pipeline output as the last stage. You can code any of the output stages in the middle of a pipeline. Each output stage processes the data in the input stream and passes the data (unmodified) to the output stream. For example, if you are debugging a problem within a pipeline, you can insert the CONS stage to display the contents of the pipeline.

Simple multiple-stage PIPE

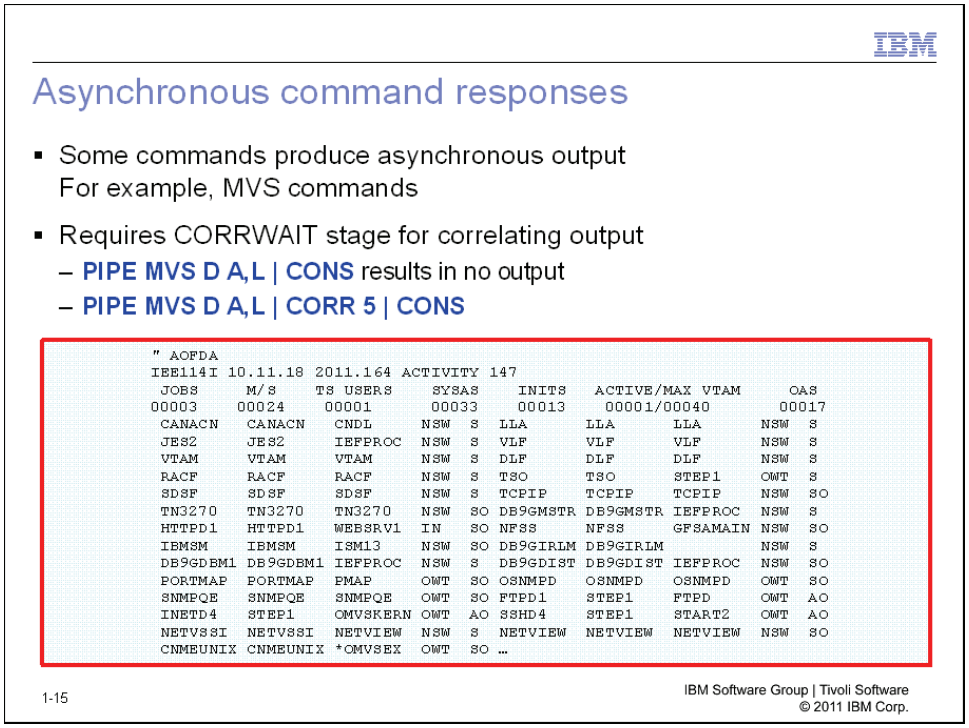


This slide shows a simple example of a pipeline with four stages:

1. Read file DSIPROFB in the NetView DSIPRF data set concatenation.
2. Remove any lines that begin with an asterisk (comments).
3. Combine the remaining single-line records into one multiline record.
4. Display the output stream on the NetView screen.

The remainder of this unit discusses each of these PIPE stages and more.

Asynchronous command responses



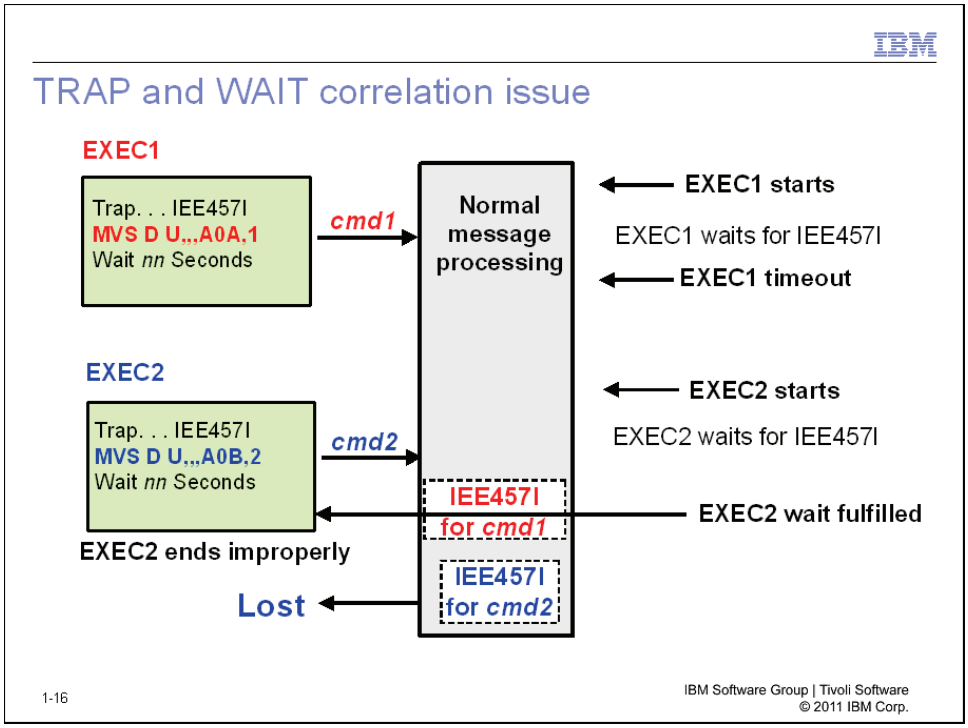
Some command responses are *asynchronous* responses. The output of the command does not return to the pipeline without the use of PIPE stages to wait for the response. An example would be a two-stage PIPE that consists of the MVS stage to issue a command and the CONSole stage to display the pipeline data. No data would result because of the asynchronous response from the MVS command.

To collect and correlate asynchronous data, the CORRWAIT (WAIT for short) stage must be specified also. It can specify a maximum time to wait for the data

Note: If you do not specify a minimum CORRWAIT time, the PIPE waits one second. If you code CORR *, the PIPE waits forever.

In most cases, you wait for the asynchronous data and specify a *terminating condition* to end the wait as soon as the condition is met. When the PIPE is in a wait state, the task that runs the PIPE also waits. No commands that are queued to the task run until the PIPE completes.

TRAP and WAIT correlation issue



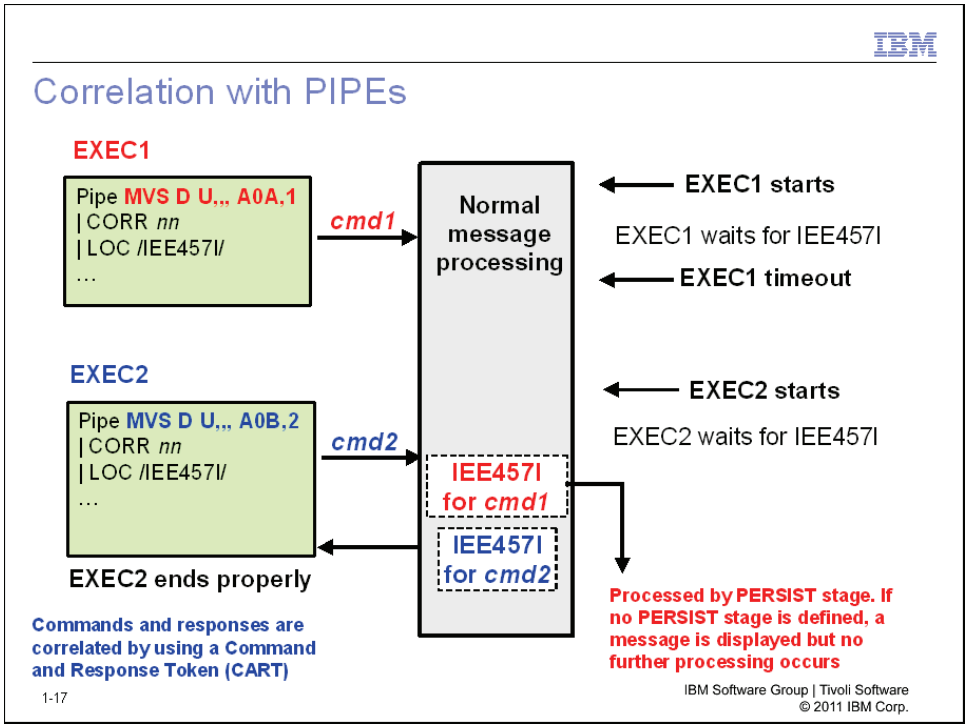
Suppose you have two EXECs that issue the MVS **D U** command. Each EXEC traps the IEE457I message in response to the command. This slide shows a scenario as follows:

1. The first EXEC, EXEC1, encounters a WAIT timeout and ends.
2. EXEC2 is issued and waits for message IEE457I.
3. When the first EE457I message arrives, it is intended for EXEC1.
4. EXEC2 is active and encounters an error because of the erroneous IEE457I message.
5. After EXEC2 ends, the IEE457I that is intended for it arrives and is then discarded.

This scenario is common when writing REXX EXECs to TRAP and WAIT for the same message ID on the same task. Two solutions are as follows:

- You can add code to the EXECs to test the message to make sure it is the correct one.
- You can use a PIPE command similar to the one shown on the next slide.

Correlation with PIPEs



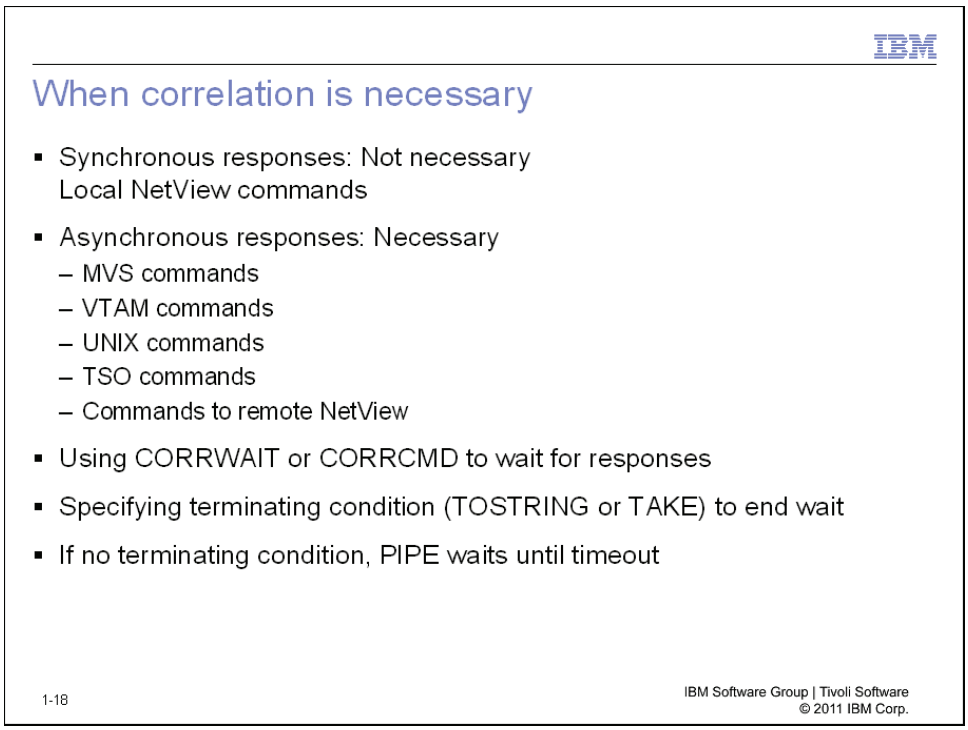
With command correlation, every command that follows the pipeline has a Command and Response Token (CART) appended to it. Only messages that match the CART route to the output stream for a command. This enables the relevant response to be identified and returned asynchronously. Only the correctly correlated replies enter the appropriate pipe.

In the example, the LOCATE stage confirms that each PIPE is waiting for the IEE457I message that pertains to the MVS command issued. The PERSIST stage can process messages that arrive after the PIPE ends. By default, the PERSIST stage displays the message on the task where the PIPE ran. Discussion about PERSIST occurs later in this unit.

Using NetView pipelines resolves two issues as follows:

- All IEE457I messages are correlated and return to the correct pipeline.
- The IEE457I that arrives after EXEC1 completes processes based on the PERSIST stage specification.

When correlation is necessary



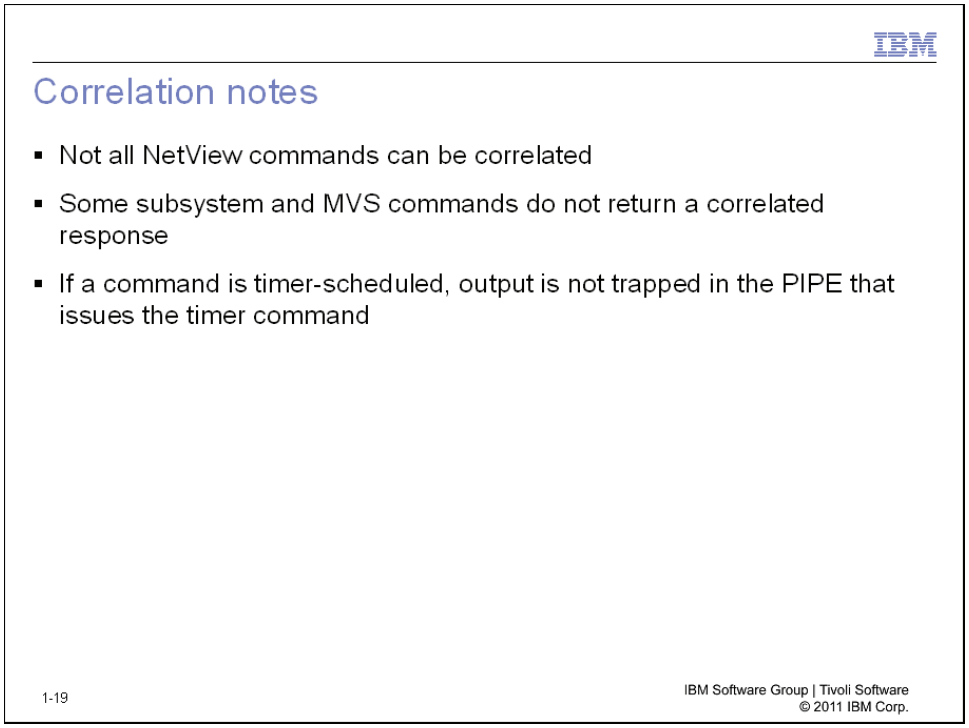
You should add correlation (CORRWAIT) to your pipelines when issuing the following types of commands:

- Commands to a remote NetView
- Most MVS commands
- VTAM commands
- TSO commands
- UNIX® commands

Note: If you do not specify a minimum CORRWAIT time, the PIPE waits one second. If you code CORR *, the PIPE waits forever.

When the PIPE is in a wait state, the task that runs the PIPE also waits. No commands that are queued to the task run until the PIPE completes.

Correlation notes

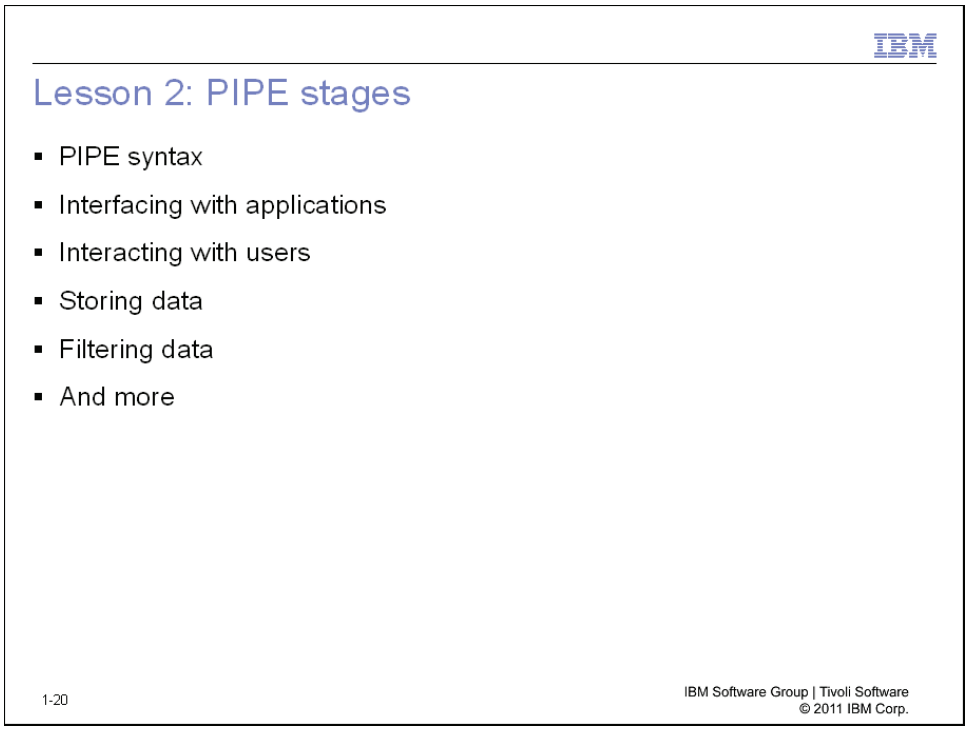


Not all commands are capable of correlation. A subset of older NetView commands are not correlatable. You can test if a command is correlated with a simple PIPE as follows:

```
PIPE NETV command_name | HOLE
```

If you see output for *command_name*, it is not correlated. Correlated command output passes to the HOLE stage and is suppressed. Discussion about the HOLE stage occurs later in this unit.

Lesson 2: PIPE stages

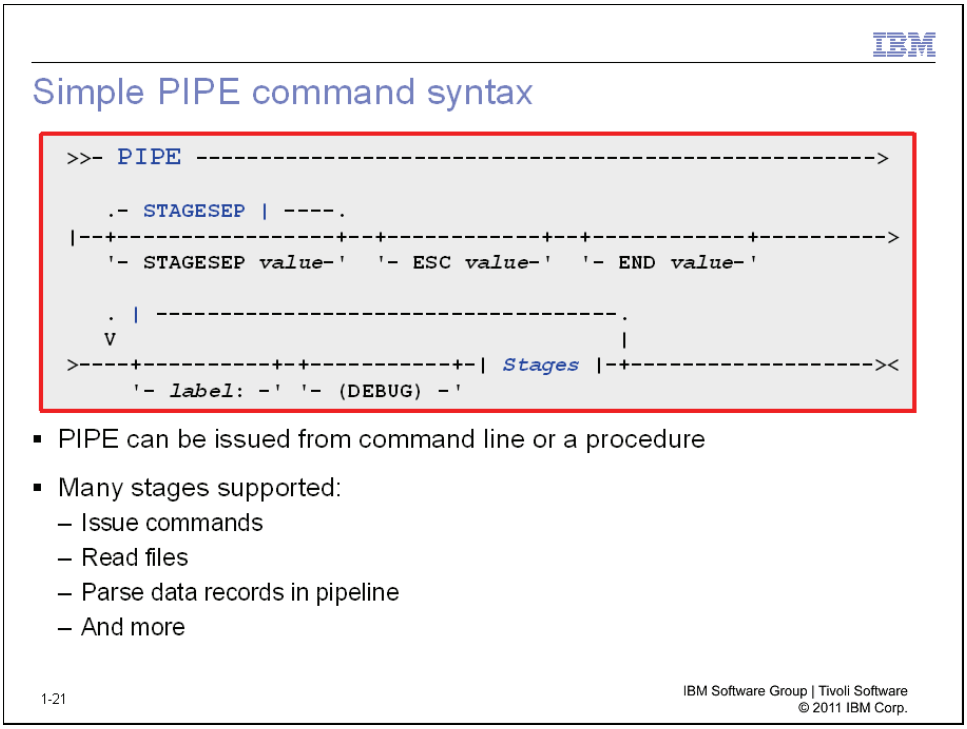


This lesson discusses the PIPE command syntax and several of the most commonly used PIPE stages.

Online help as follows is available when you are logged on to NetView:

- **HELP PIPE SYNTAX**: Displays help for the syntax of the PIPE command.
- **HELP PIPE STAGES**: Displays a list of PIPE stages. Select a stage in the list to see the help.
- **HELP PIPE *stage_name*:** Displays the help for the specified stage. For example, HELP PIPE NETVIEW displays the help for the NETVIEW stage.

Simple PIPE command syntax



This slide shows the basic PIPE command syntax. PIPE stages are separated by a character called a *stage separator*. For example, **PIPE NETVIEW LIST DSILOG | LOC ... | ... | CONSOLE** uses the default stage separator as follows:

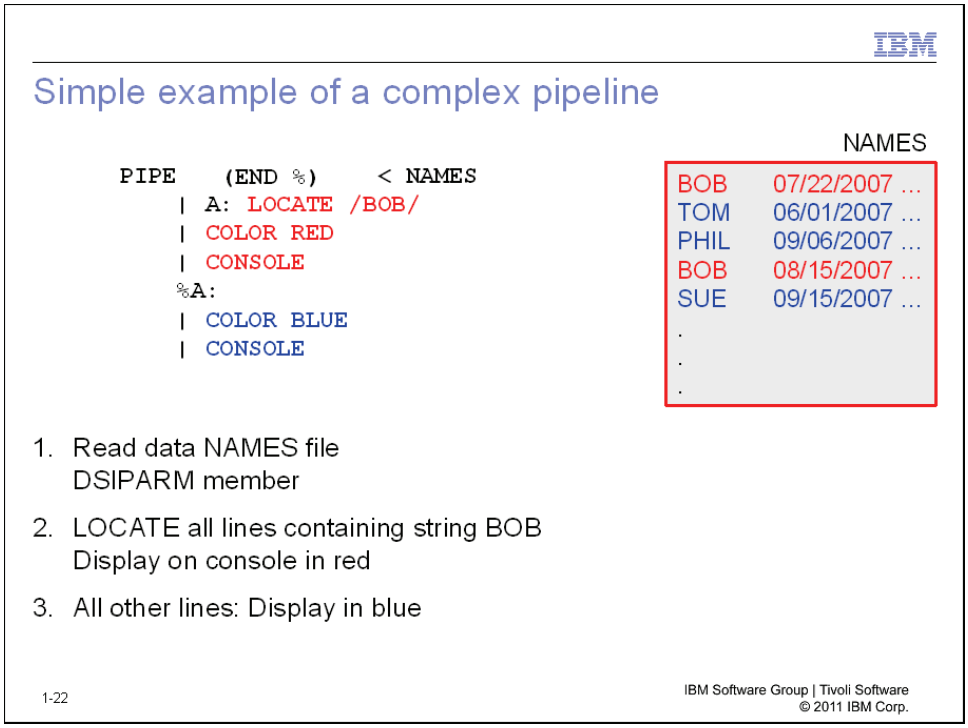
- With the NETVIEW stage to issue a LIST DSILOG command
- With the LOC stage to filter data
- With the CONSOLE stage to display the results

You can use the **STAGESEP** option to specify the character to use for separating the stages within a pipeline. The default stage separator is the character x'4F'. Depending on the code page used by your workstation, this stage separator is a solid vertical bar (|), split vertical bar (¦), or exclamation mark (!).

The **ESC** option indicates that the character that follow the specified character is treated literally when the pipeline specification is parsed. Alternatively, you can use the stage separator character to self-escape itself. Two side-by-side separators resolve to one such character taken literally.

The **END** option identifies a character that is used for signifying the end of a pipeline.

Simple example of a complex pipeline



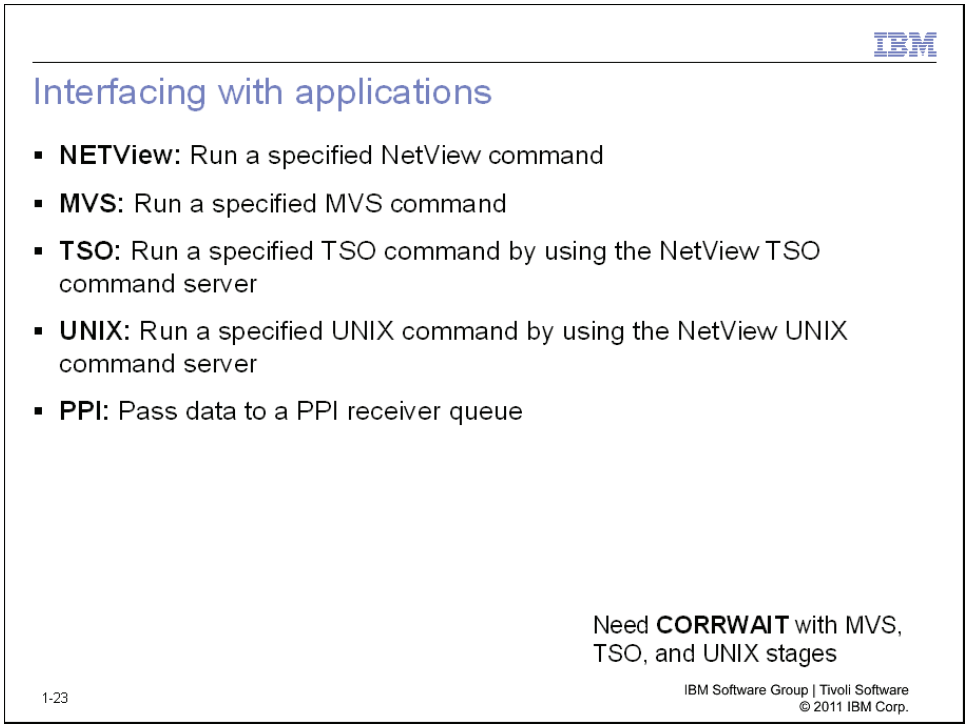
This slide shows a simple example of a *complex pipeline*, a series of simple pipelines that are connected with labels. Some stages accept multiple input streams and others generate multiple output streams. In most cases, when you see an END character that is defined in a pipeline, it is a complex pipeline.

Starting with a simple pipeline, the NAMES file is read (from the NetView DSIPARM data set, by default) with the **PIPE <** stage. The **LOCATE** stage filters out all records that do contain the string BOB and displays them in red on the NetView operator screen.

Using the END character (%), you can specify the end of a pipeline. All output that the first pipeline filters passes to the second pipeline. The NetView operator screen displays the data that is in the second pipeline in blue.

Discussion about complex pipelines occurs later in this unit.

Interfacing with applications

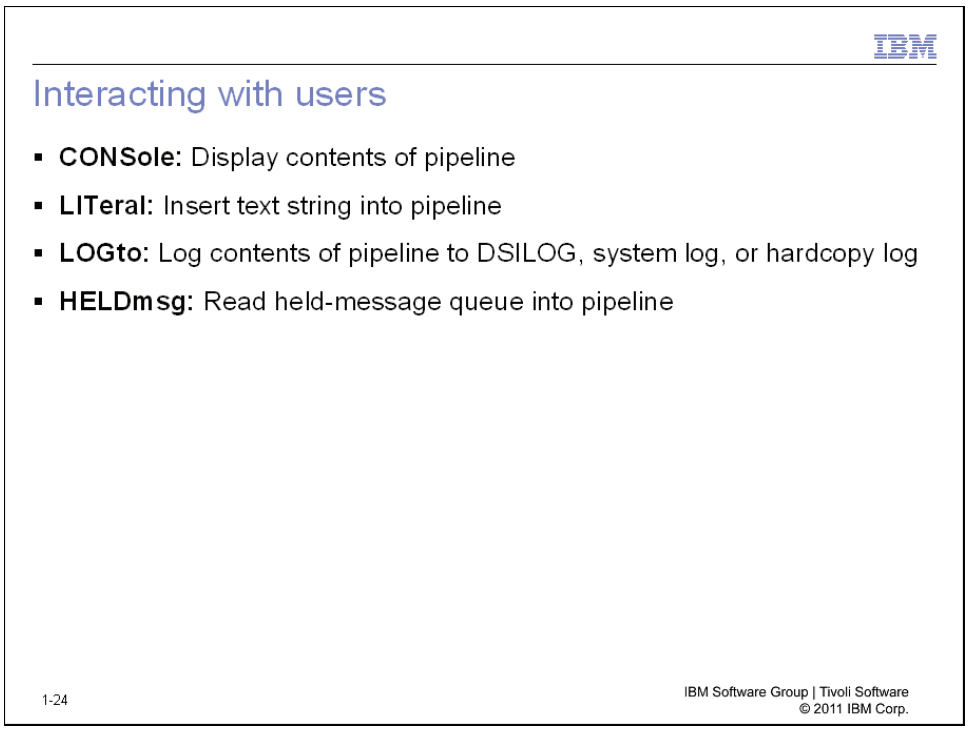


Using NetView pipelines, you can issue NetView, MVS, VTAM, TSO, or UNIX commands as follows:

- **NETVIEW stage:** Specifies to run a NetView, MVS, or local VTAM command. An example is PIPE NETV LIST DSILOG.
- **MVS stage:** Specifies to run an MVS command. The MVS stage can be used instead of coding MVS commands within the NETVIEW stage. An example is PIPE MVS D A,L.
- **TSO stage:** Routes a command to a NetView TSO server, which is a batch job submitted by NetView or a started task.
- **UNIX stage:** Routes a command to a NetView UNIX server where the command is run and the results returned.

You can also use the **PPI** stage to send and receive data from non-NetView applications by using the NetView program-to-program interface (PPI).

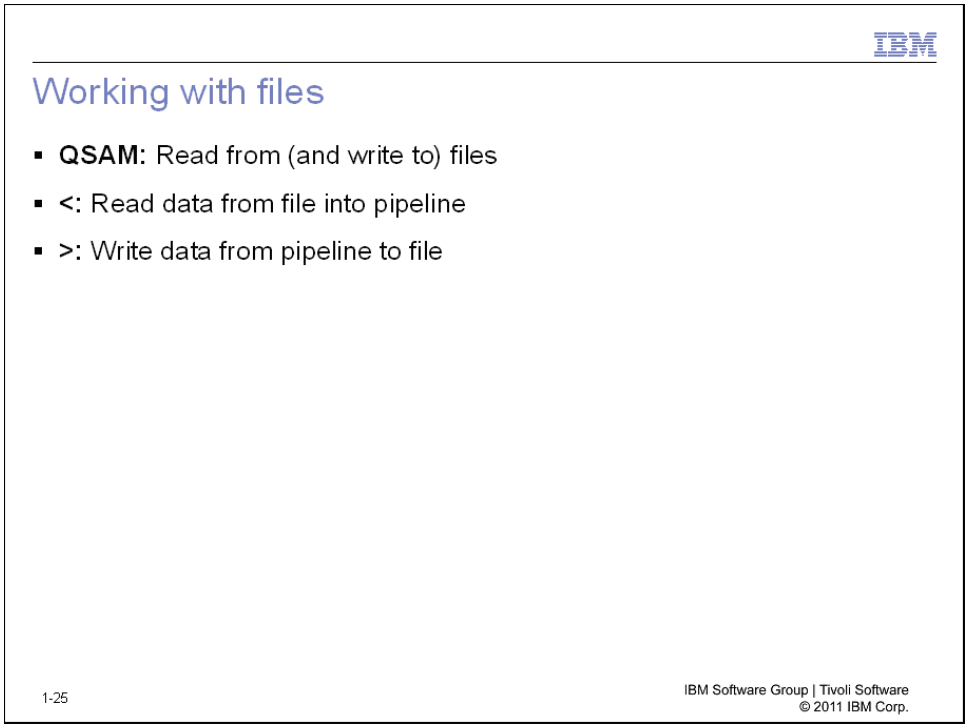
Interacting with users



This slide shows the pipeline stages that can be used for interacting with an operator. For example, you can display a message to the operator with the CONSOLE stage.

Note: If you specify CONSOLE ONLY, messages are to be displayed but not held, logged, or exposed to automation. For example, use the ONLY option when sending text to the operator.

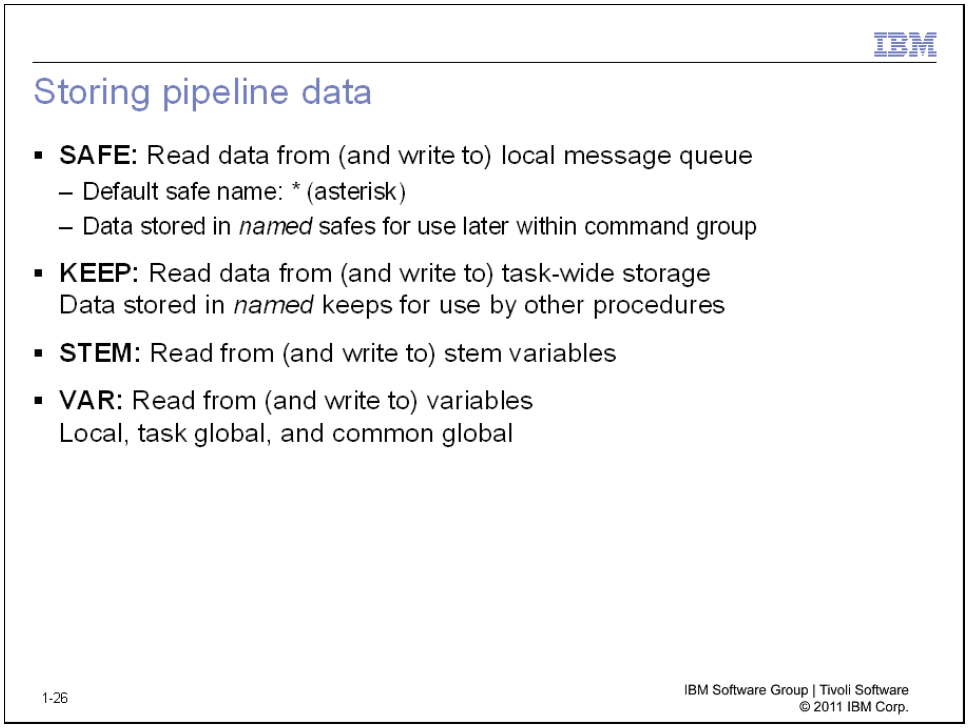
Working with files



This slide shows the pipeline stages that can be used for reading from and writing to members of partitioned data sets.

The members can also be loaded in NetView storage for quicker access by programs with the INSTORE stage. The members are then read from storage instead of from the disk when you use the QSAM or < stage. The in-storage members are also viewable when operators use the BROWSE command to view a member.

Storing pipeline data



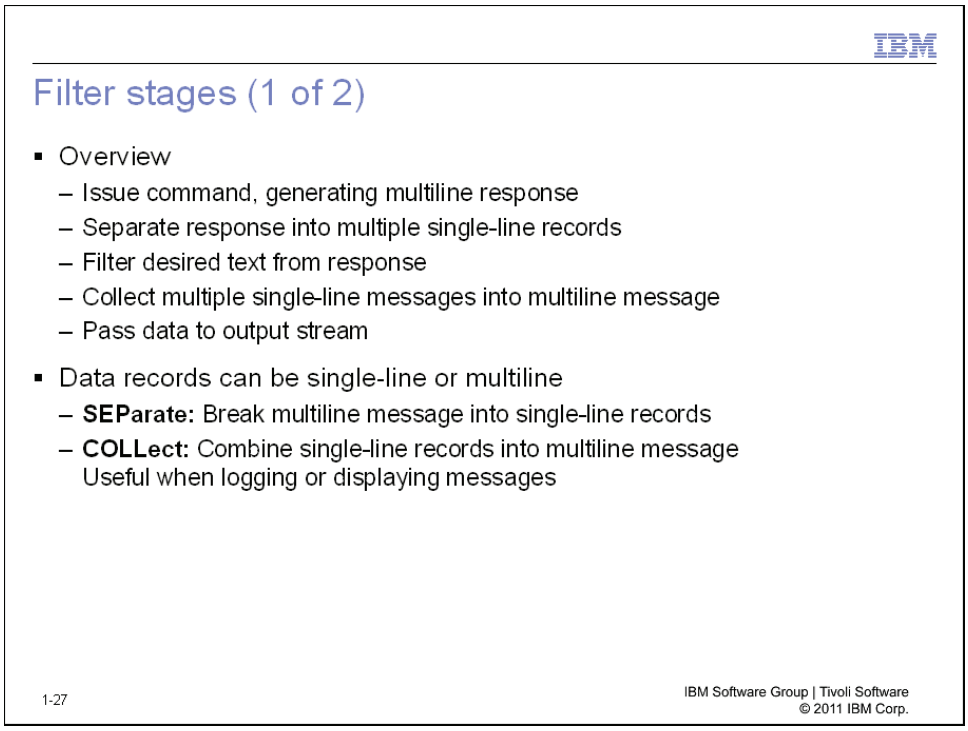
This slide shows the pipeline stages that can be used for storing data. Data can be stored internally in *safes* or *keeps* or in variables and arrays.

A common pipeline practice is using REXX EXECs to issue commands (for example, NetView LIST DSILOG). The EXEC filters the pipeline contents, and stores the output stream in a stem variable. The EXEC can loop through the contents of the stem variable and parse to determine the log that was active, for example.

Note: You can use PIPE EDIT to perform the same logic without calling a REXX procedure.

Note: You cannot issue these stages from the NetView command line.

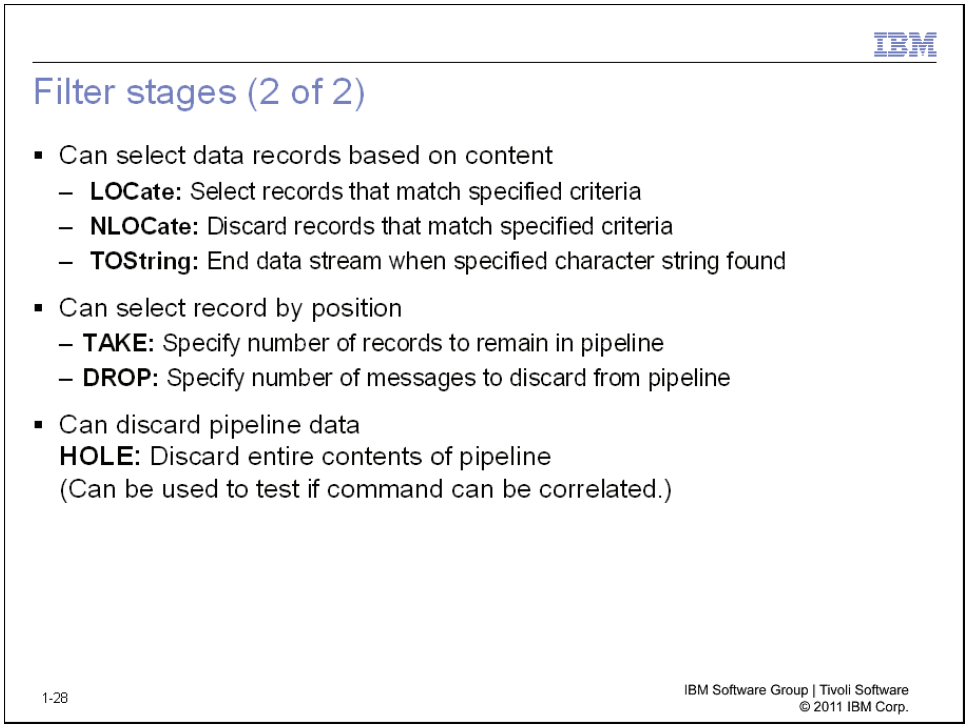
Filter stages (1 of 2)



Using pipeline filters for working with data, you can accomplish the following tasks:

- Separate multiline messages into multiple single-line messages (SEPARATE)
- Collect multiple single-line messages into a multiline message (COLLECT)

Filter stages (2 of 2)

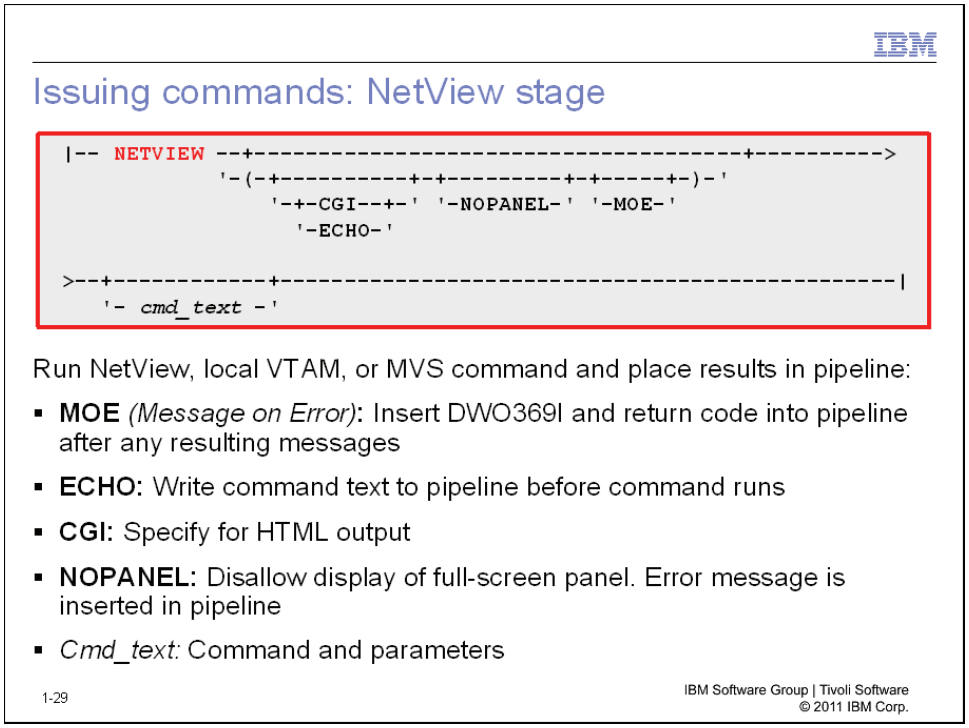


Several stages select pipeline messages to include or discard. They read all messages in the pipeline, but write only those that meet selection criteria to the stage output stream. Using selection filters, you can accomplish the following tasks:

- Keep all messages that contain a match for a specified text string (LOCATE).
- Discard all messages that contain a match for a specified text string (NLOCATE).
- Keep messages up to and including the message that contains a match for a specified text string (TOSTRING).
- Keep the first or last *nn* messages in the pipeline (TAKE).
- Discard the first or last *nn* messages from the pipeline (DROP).
- Empty the pipeline of messages after writing them to a variable or testing a command to see if it is correlated (HOLE).

TOSTRING, TAKE, and DROP stages generate a terminating condition for the CORRWAIT stage.

Issuing commands: NETView stage



The **NETVIEW** stage (abbreviated as NETV) runs a NetView, MVS, or local VTAM command. Use the NETV stage to issue VTAM DISPLAY, VARY, and MODIFY commands in a local or remote domain. The resulting messages are placed in the pipeline.

Message **o**n **e**rror (MOE) examines the return code from the command. If the return code is not zero, MOE inserts message DWO369I (containing the return code) into the stream after any messages that the command generates. If you do not specify MOE, return codes from commands are ignored.

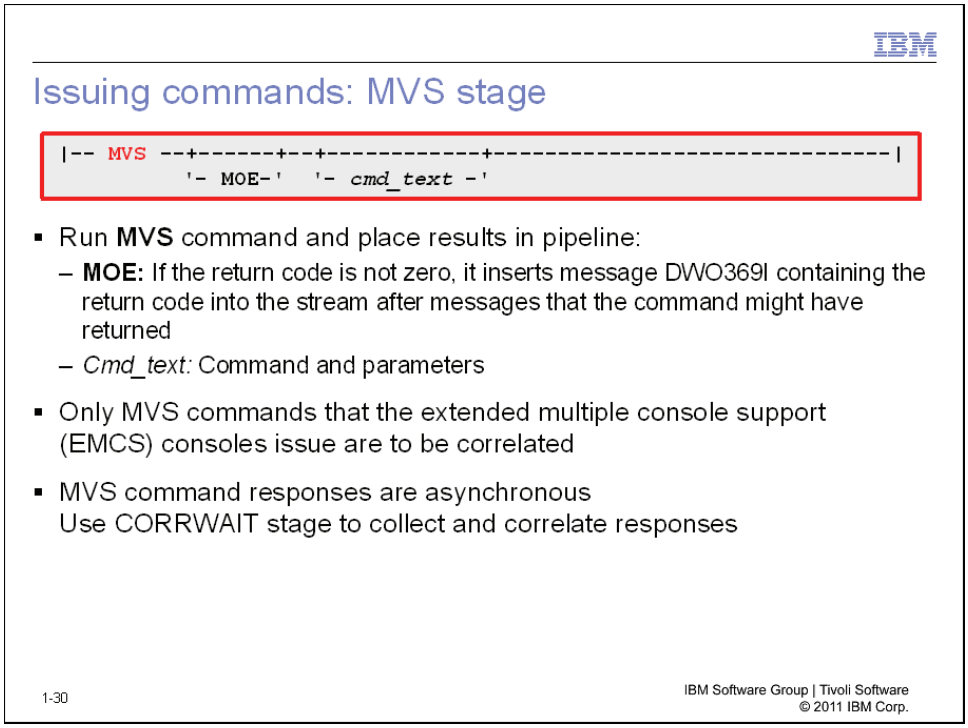
When NOPANEL is specified, a full-screen panel display is disallowed. If a full-screen panel is encountered, message BNH113W is inserted into the pipeline and the command receives an error code.

You can place the NETVIEW stage anywhere in the pipeline specification. If NETVIEW is not the first stage, the command text is optional. The command runs once for each message that the previous stage delivers. Every time the command runs, the input message becomes the current message during the processing.

The NETVIEW stage does not require an output stream, meaning that NETVIEW can be a last stage. Also, if a stage that follows NETVIEW disconnects, the NETVIEW stage continues to process if it has an input stream.

Asynchronous responses are a subset of NetView command responses. To collect and correlate asynchronous data, the CORRWAIT (WAIT for short) stage must follow.

Issuing commands: MVS stage

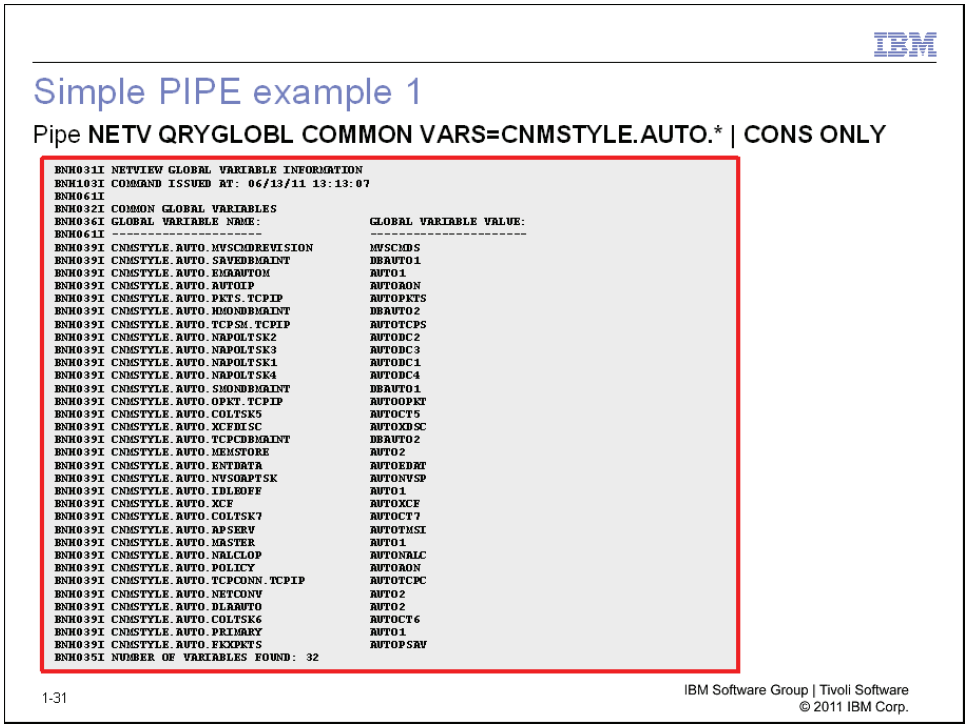


The **MVS** stage specifies an MVS command to run. You can use the MVS stage instead of coding MVS commands within the NETVIEW stage. All rules that apply to an MVS command that is issued by using the NETVIEW stage also apply here. Any restrictions that apply to running an MVS command under PIPE NETVIEW also apply to PIPE MVS.

MVS system commands can be correlated only when issued from extended multiple console support (EMCS) consoles.

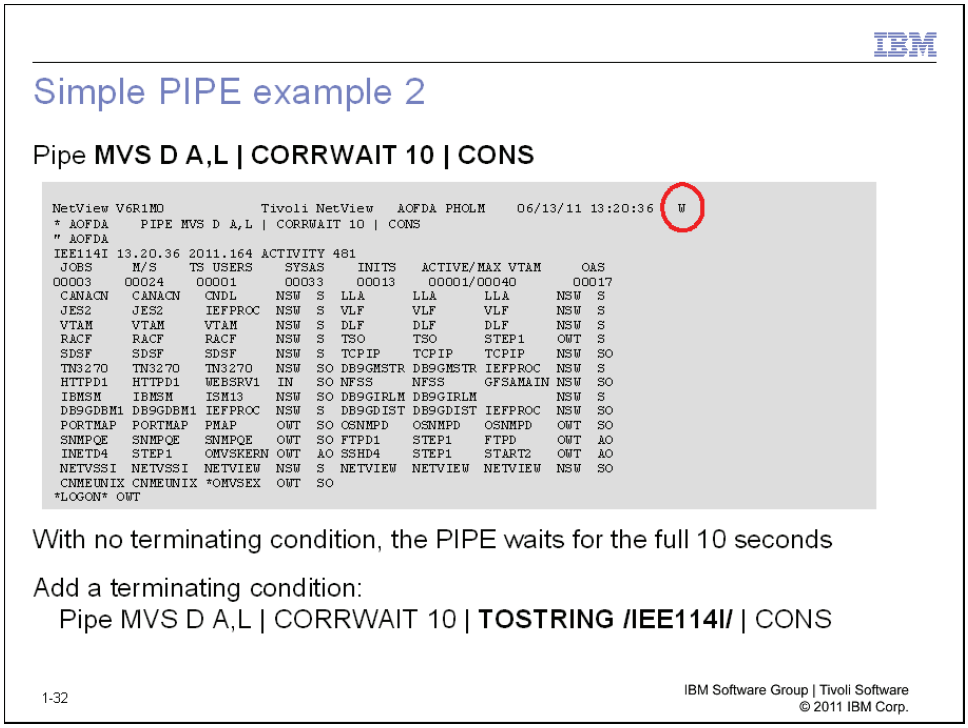
MVS command responses are *asynchronous*. To collect and correlate asynchronous data, the CORRWAIT (WAIT for short) stage must follow.

Simple PIPE example 1



The first example is a simple pipeline that issues the **QRYGLOBL** command to query all NetView common global variables that begin with the *CNMSTYLE.AUTO.* string. The output is to be displayed to only the console. Data in the pipeline can be filtered with LOC, NLOC, TAKE, DROP, and so on.

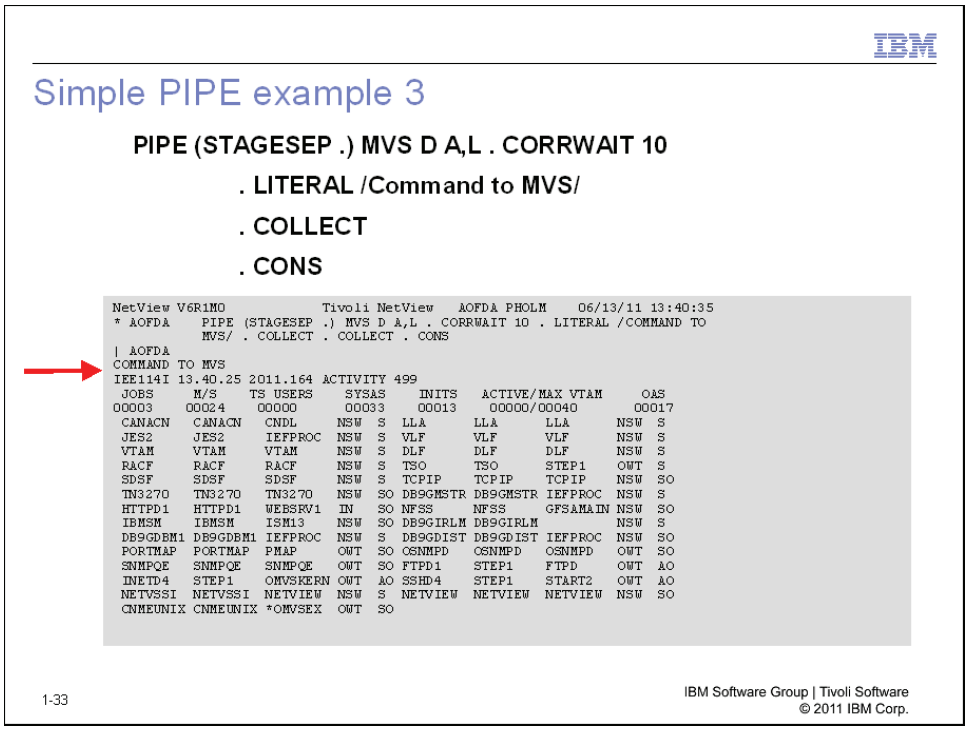
Simple PIPE example 2



This example issues the **D A,L** command by using the **MVS** stage. MVS command responses are *asynchronous* and require the addition of the **CORRWAIT** stage to wait (in this case, 10 seconds) for all responses. The IEE114I multiline message is to be displayed as soon as it is received, but you wait for 10 seconds for any additional messages.

To end the wait when the IEE114I arrives, use a PIPE stage to generate a *terminating condition*. For example, use TAKE 1 or TOSTRING /IEE114I/. If you do not use the CORRWAIT stage, you do not see a response.

Simple PIPE example 3

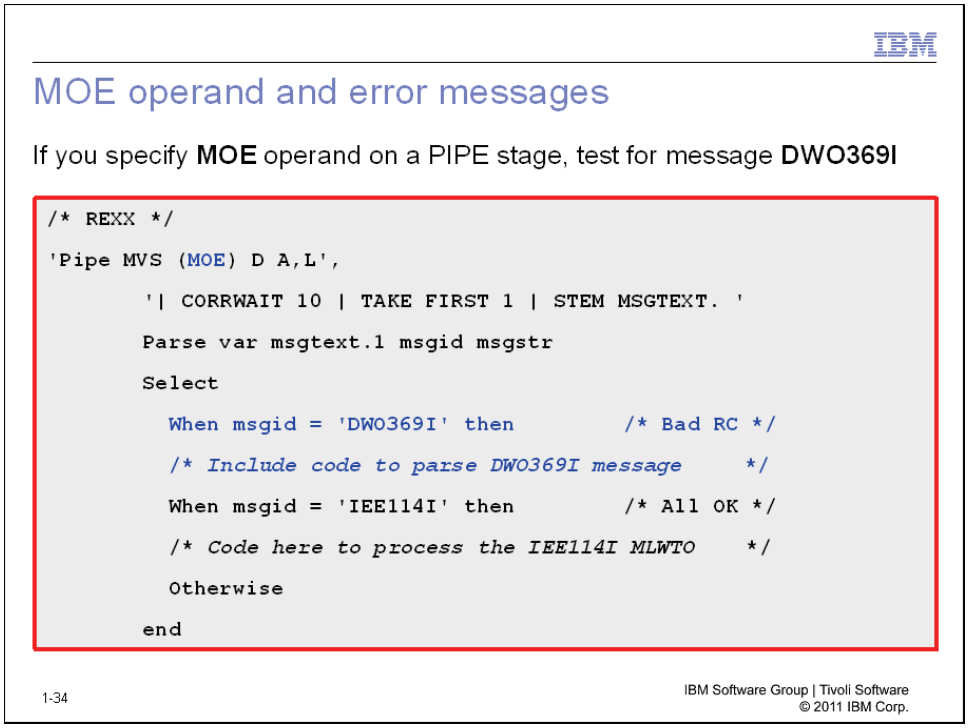


This example issues the **D A,L** command by using the **MVS** stage with a *stage separator* of a period (.). The **CORRWAIT** stage is used for waiting for the asynchronous response.

The **LITERAL** stage inserts a text string into the pipeline when the CORRWAIT stage ends. Collect text string and IEE114I into a multiline message by using the **COLLECT** stage. There is no terminating condition. The output is not displayed until the CORRWAIT ends.

The text string is displayed first. Data that appends in the pipeline output displays as *last in, first out* (LIFO), by default. You can change the order by using the **REVERSE** stage.

MOE operand and error messages

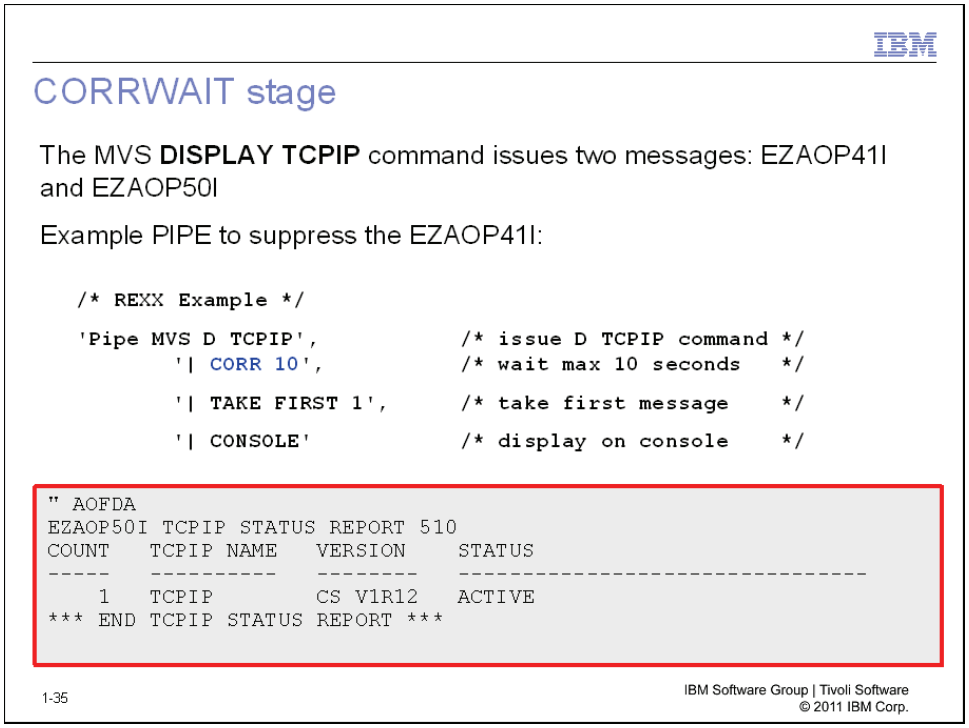


The text of a DWO369I resembles the following text:

```
DWO369I  stagename STAGE (stagenum) HAD RETURN CODE retcode
```

Your procedure can display the DWO369I to users to inform them of the error.

CORRWAIT stage



The **CORRWAIT** stage allows asynchronous messages (generated by the previous stage) to be returned to the pipeline, and sets a specified time limit if needed. Asynchronous messages are those that return to the NetView program from commands running in another application.: Example commands are MVS, VTAM, TSO, UNIX, and commands running at another NetView. When asynchronous responses are to result from a command issued in a pipe, the next stage command must be CORRWAIT. Without CORRWAIT, the messages are lost.

If a message arrives after the time interval expires, it is discarded. In some instances, a very large time interval might be needed. The wait time can be long between a message arrival and time interval expiration.

A *terminating condition* to end the wait state should follow a CORRWAIT stage. **TAKE** and **TOSTRING** are example stages that end the wait. Each causes the **CORRWAIT** stage to disconnect. CORRWAIT without a *terminating condition* (for example TAKE) waits until the timeout value for all messages.

Note: LOCATE does not cause a terminating condition.

In example on the slide, the **MVS DISPLAY TCPIP** command is issued with a CORRWAIT of 10 seconds. The **TAKE FIRST 1** defines a terminating condition that takes control when the first message arrives (if fewer than 10 seconds). Discussion of the TAKE stage occurs later in this unit.

In this example, the second message, EZAOP41I, message is suppressed:

```
EZAOP41I 'DISPLAY TCPIP' COMMAND COMPLETED SUCCESSFULLY
```

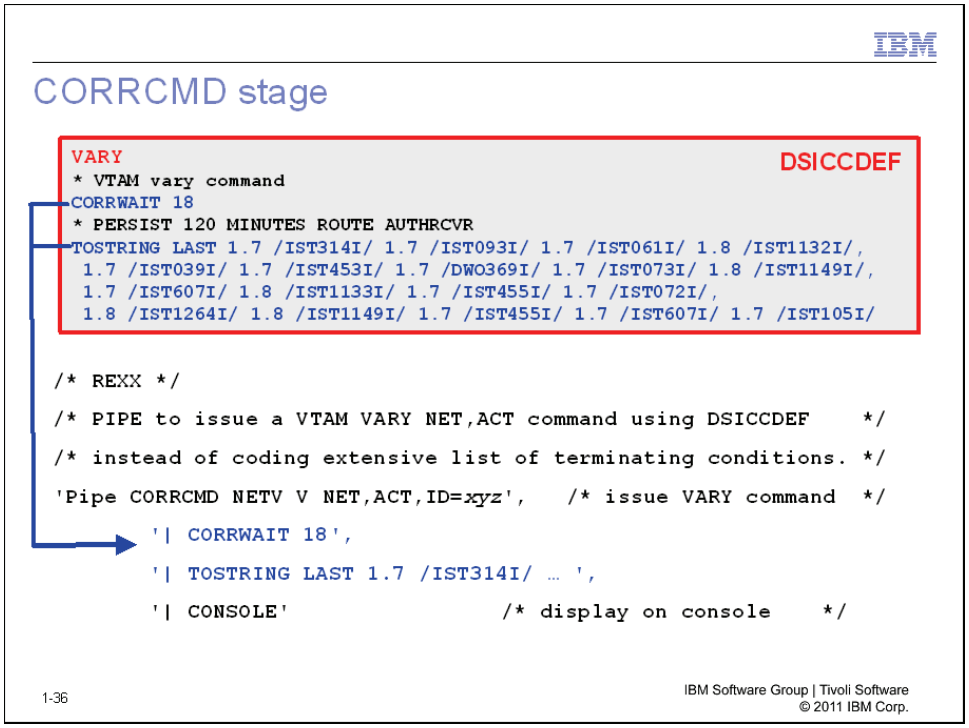
CORRWAIT can end under the following conditions:

- Operator issues GO command.
- PIPE issues GO command.
- Match is found for expected condition in a filter stage, for example, TOSTRING or TAKE stages.
- CORRWAIT stage timeout occurs.
- PIPEND stage occurs.
- Secondary input stream disconnects.
- Message is received.

The wait indicator (**W**) is displayed on the operator screen when CORRWAIT waits for messages.

Note: When the PIPE is in a wait state, the task that runs the PIPE also waits. Commands queued to the task do not run until the PIPE completes.

CORRCMD stage



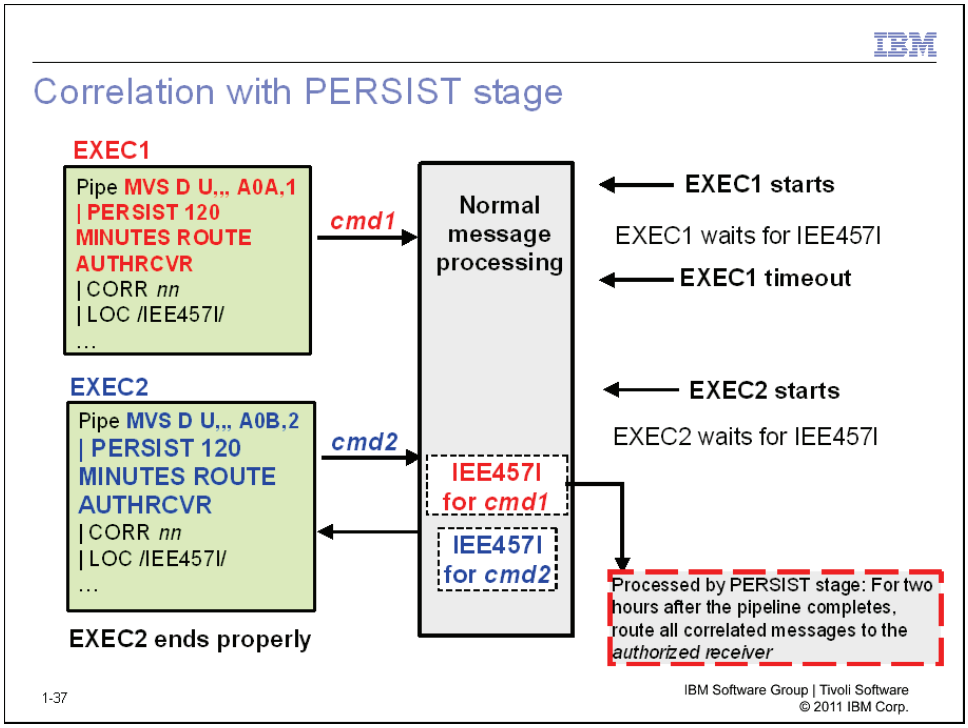
The **CORRCMD** stage (abbreviated as **CC**) processes a command and inserts appropriate correlation wait and termination condition stages. It gathers data from the command being processed. CORRWAIT is included in the inserted stages.

The timeout interval and termination conditions to use are defined in NetView DSIPARM member **DSICCDEF**. A sample file named DSICCDEF (CNMS1082) is provided with NetView. This member contains several commands with appropriate timeout and termination stages. Issue HELP CCDEF or CCDEF QUERY=* NetView commands to display information pertaining to these settings.

This slide shows how to use the sample definitions defined in DSICCDEF for the VTAM VARY command. DSICCDE defines the CORRWAIT and TOSTRING stages. When a PIPE CORRCMD stage is encountered for a VTAM VARY command, the CORRWAIT and TOSTRING stages append into the pipeline.

To include the PERSIST stage, uncomment the **PERSIST 120 MINUTES ROUTE AUTHRCVR** statement. To simplify the coding of your pipeline stages, use DSICCDEF and CORRCMD

Correlation with PERSIST stage



The **PERSIST** stage defines the action to perform when messages arrive after termination of the correlation that a CORRWAIT stage represents. The conditions that PERSIST defines are enabled after termination of the preceding CORRWAIT stage (WAIT) when one of these conditions is true:

- WAIT times out.
- WAIT responds to GO or RESET.
- WAIT ends prematurely because of a PIPEND (non-zero return code).
- DEBUG option occurs for the PERSIST stage. The PERSIST stage activates when the DEBUG option is specified.

If you specify more than one PERSIST stage for a command correlation environment, only the last specified PERSIST stage takes effect.

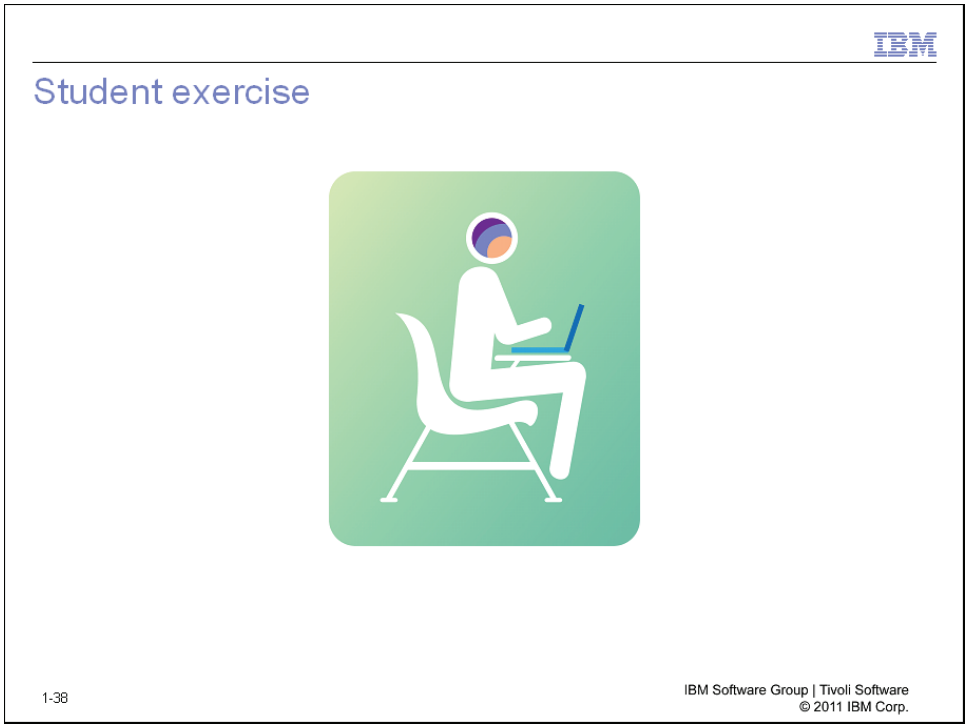
Messages subject to the DISPLAY action are exposed to user exits, trapping, automation, and logging. For more information about exposure, see “PIPE EXPOSE stage” on page 1-101.

Use the **LIST PERSIST** command to display the status of all enabled PIPE PERSIST elements or use the STOP PERSIST command to end an enabled persist.

Messages subject to ROUTE action are routed first, then are exposed as for other messages.

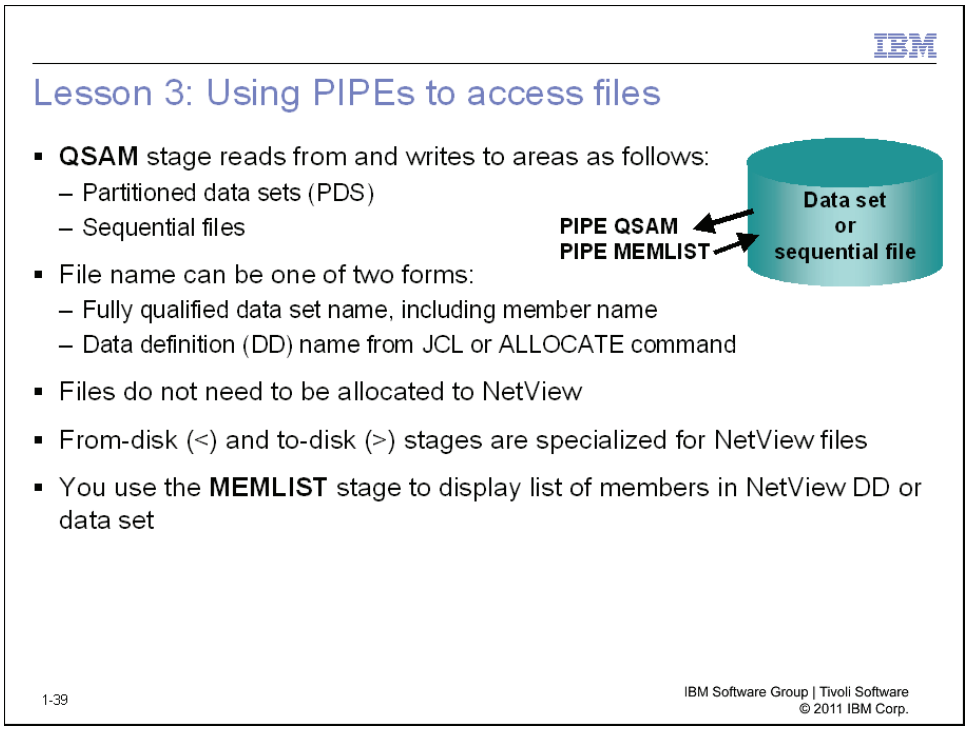
A message subject to COMMAND action is provided as the current message when the indicated command runs. Any output from the command, including the original message, is subject to exposure in the same way as the output of a command issued from the command facility command line. When PERSIST invokes a command, it does so with the same authority as was in effect for the pipeline which established the PERSIST action.

Student exercise



Open your *Student Exercises* book and perform Exercises 1 and 2.

Lesson 3: Using PIPEs to access files



The **QSAM** stage reads and writes from dynamically allocated data definition (DD) names or data sets. Other devices are also supported when allocated for physical sequential access. The < and > (synonyms for QSAM read and write) can be used only when < and > are immediately followed by a data set name enclosed in quotation marks.

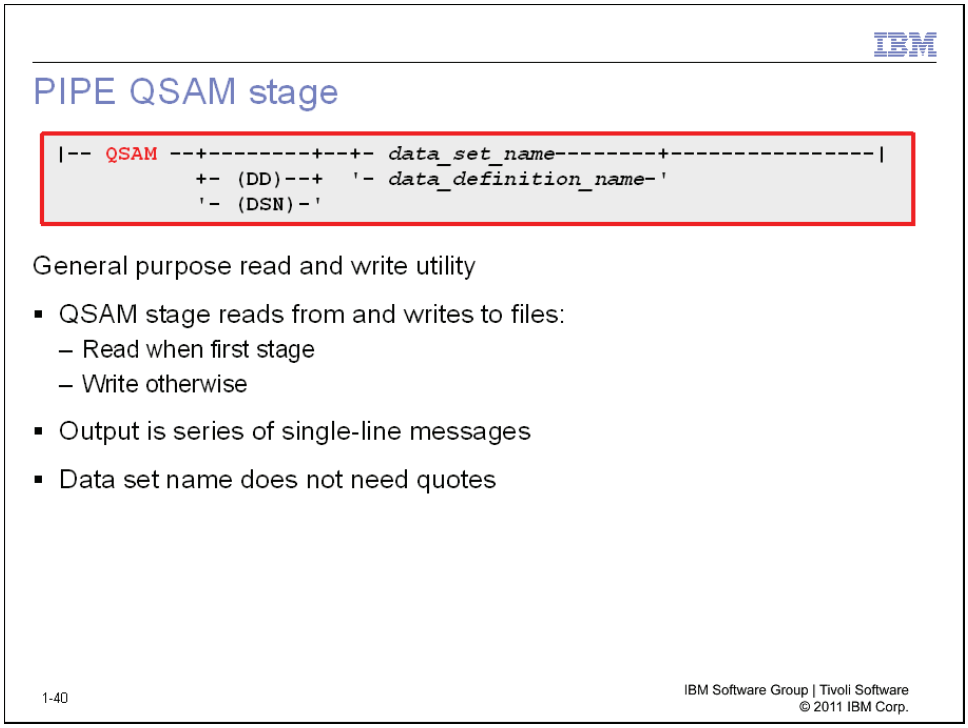
When QSAM is the first stage in a pipeline, the file is read into the pipeline. Otherwise, the contents of the pipeline are written to the file.

The < (from-disk) stage must be the first stage of pipeline. The from-disk stage is limited to a subset of data sets that are allocated to NetView. Issue **BROWSE !** to display the list.

Note: For NLS users, you can use X'5A' to represent the exclamation point character (!).

The **MEMLIST** stage creates a list of members in one or more partitioned data sets (PDS) or data definitions (DD). For a DD, the members are listed for each data set in the concatenation. Members that are defined in an operator data set and members that are defined by using INSTORE COMMON are also listed.

PIPE QSAM stage



You can use the **QSAM** stage with either a data definition (DD) name that is defined by the ALLOCATE command, or a fully qualified data set name. You can enclose a data set name in single quotes. The quotes are ignored.

When specified as a first stage, QSAM reads from the data definition name or data set. When not specified as a first stage, QSAM writes to the data definition name or data set. The messages received on the input stream pass to the output stream.

PIPE < and PIPE > stages

These stages are specialized versions of PIPE QSAM for NetView files. The default data set name is DSIPARM. Additional parameters are supported. INCL reads all included members and resolves any *Data REXX* statements. DISKONLY reads members from disk only. In some cases, files are readable into NetView storage. DISKONLY does not read the in-storage copy.

When you use a data_definition_name of *, NetView searches all standard DD names, if allocated, for the specified member name in the following order:

1. DSICLD
2. DSIPARM
3. DSIPRF
4. DSIVTAM
5. DSIMSG

```
6. CNMPNL1
7. BNJPNL1
8. BNJPNL2
9. DSILIST
10. DSIOPEN
11. DSIASRC
12. DSIARPT
```

Issue **BROWSE !** to display the list of supported standard DD names. For NLS users, you can use X'5A' to represent the exclamation point character (!).

Security considerations

Accomplish security for PIPE QSAM, <, and > stages by implementing security for **READSEC** and **WRITESEC** commands. For example, operator NETOP1 can read from DSIPARM but is not permitted to write to DSIPARM. Implement the stage as follows:

- Issue the READSEC DD=DSIPARM to test read access:

```
DSI633I READSEC COMMAND SUCCESSFULLY COMPLETED
```

- Issue the WRITESEC DD=DSIPARM to test write access:

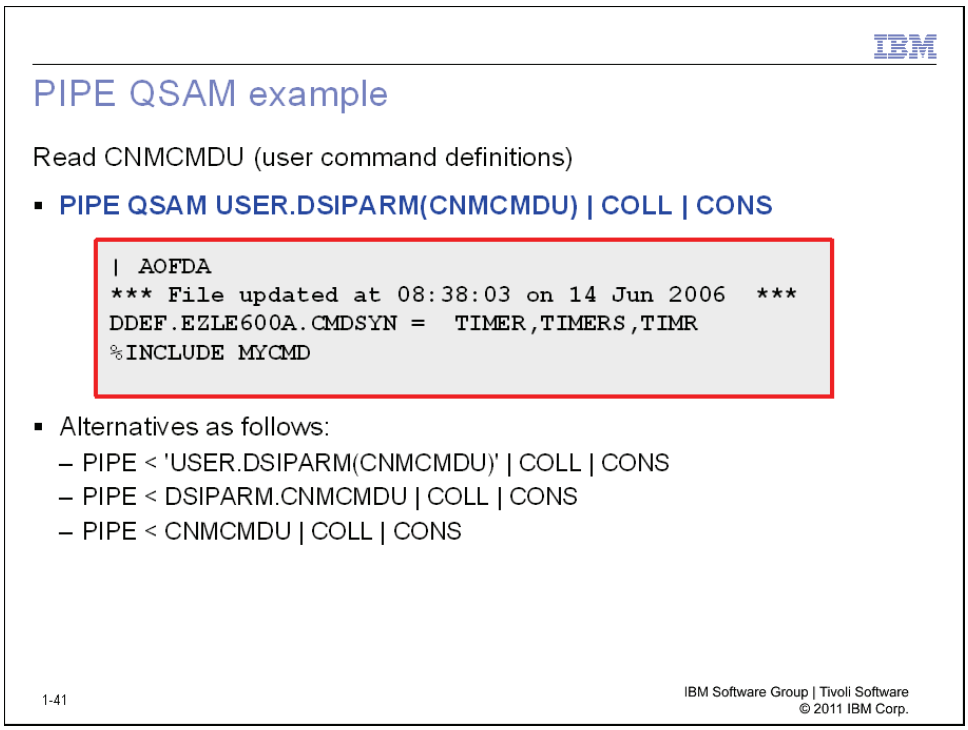
```
BNH234E 'NETOP1' IS NOT AUTHORIZED TO USE KEYWORD 'DSIPARM'
BNH235E THE KEYWORD 'DSIPARM' IS PROTECTED BY COMMAND IDENTIFIER
'*.*.WRITESEC.*' IN 'TBLNAME=SECQSAM'
DSI213I ACCESS TO 'DSIPARM' IS NOT AUTHORIZED
```

Example Command Authorization Table (CAT) statements for this example are as follows:

```
* Define oper group
GROUP   NVOPS1   NETOP1,NETOP2,OPER1,OPER2,OPER3
* Restrict access to READSEC
PROTECT          *.*.READSEC.DSIPARM.*
* Allow NVOPS1 to READSEC
PERMIT  NVOPS1   *.*.READSEC.DSIPARM.*

* Restrict access to WRITESEC
PROTECT          *.*.WRITESEC.*
PROTECT          *.*.WRITESEC.*.*
* No PERMIT for WRITESEC = no writes allowed
```

PIPE QSAM example



In the following example, the pipe reads member CNMCMDU explicitly from the USER.DSIPARM data set. An included member is MYCMD, but the QSAM PIPE displays only the member name.

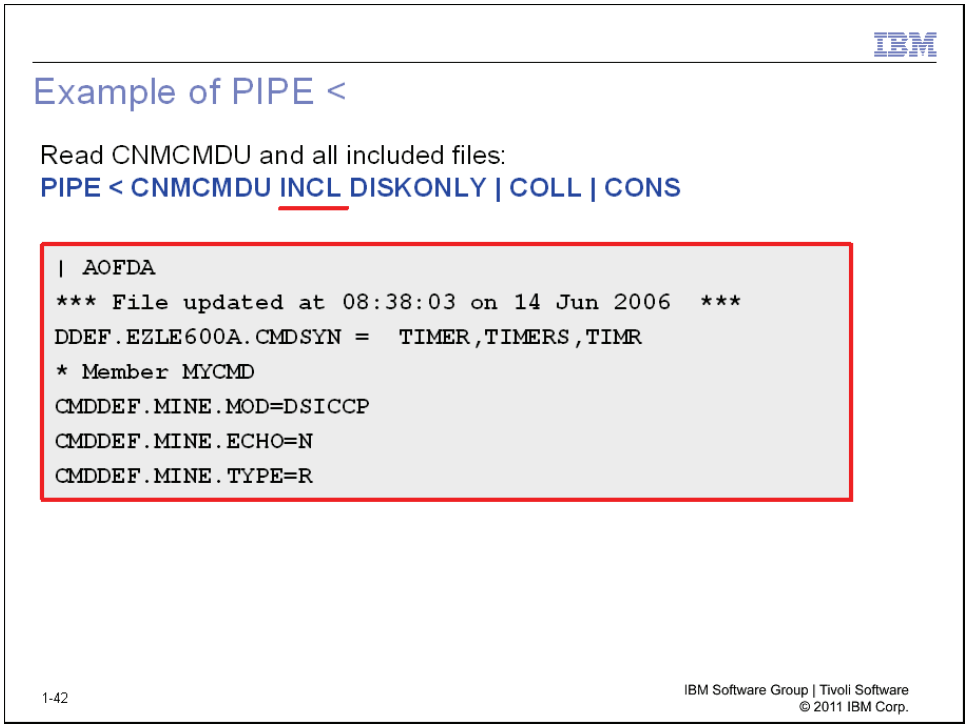
```
PIPE QSAM USER.DSIPARM(CNMCMDU) | COLL | CONS
```

For comparison, this is a section of a simple REXX EXEC to read members.

```
'TRAP AND SUPPRESS ONLY MESSAGES *'        /* TRAP/SUPPRESS MSGS  */
'ALLOCATE DA('DataSetName') SHR FREE'      /* ALLOC/CONNECT FILE  */
'WAIT FOR MESSAGES'                        /* WAIT FOR MESSAGES   */
'MSGREAD'                                  /* READ A MESSAGE IN   */
'TRAP NO MESSAGES'                         /* DISABLE TRAP MSGS   */
/* CNM272I  ddname IS NOW [ALLOCATED | DEALLOCATED]                */
If MSGID() = 'CNM272I' then                /* allocate worked?    */
  DO                                       /* PROCESS CNM272I MSG */
    DDNAME = MSGVAR(1)                     /* SAVE DYNAMIC DDNAME */
    ADDRESS MVS 'EXECIO * DISKR' DDNAME '(STEM Output.'
    Say 'File' DataSetName'/'ddname 'has' output.0 'lines:'
    Do i=1 to Output.0                     /* Output loop         */
      SAY SUBSTR(Output.i,1,68)            /* DISPLAY LINE TO USER */
    End                                    /* Output loop         */
    Say 'TYPE EXEC is now finished'
  END                                      /* PROCESS CNM272I MSG */
ELSE                                       /* MSG IS CNM272I      */
  SAY MSGID() MSGSTR()                     /* DISPLAY MESSAGE     */
```

You can use one PIPE statement to replace this REXX program.

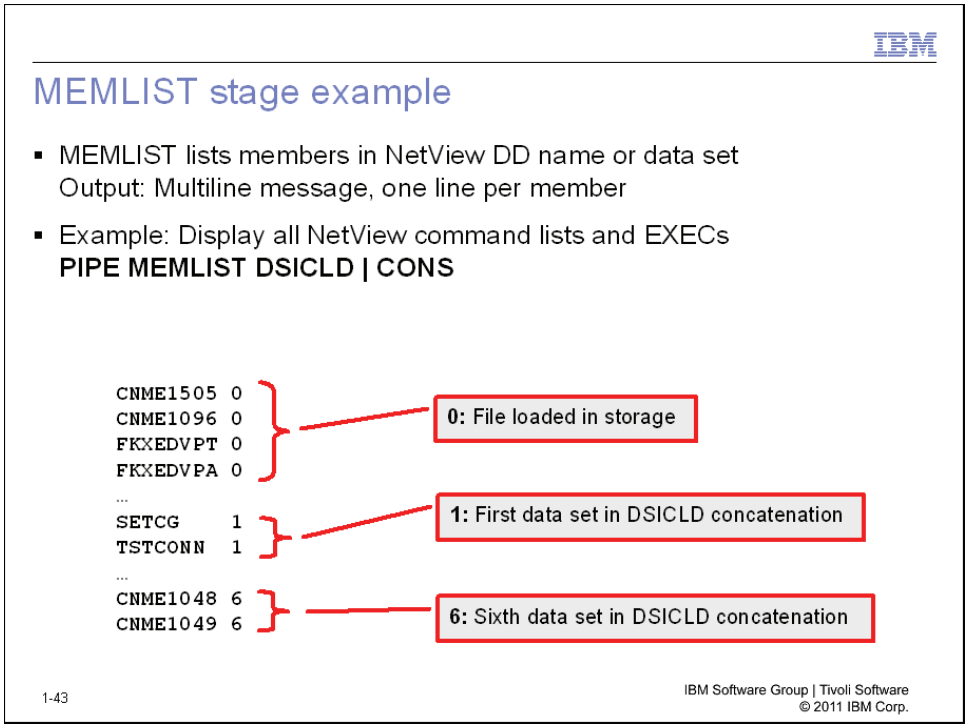
Example of PIPE <



In the example, the pipe reads CNMCMDU, using the from-disk stage. The standard NetView DD names are searched and the member found in the DSIPARM library concatenation. The included member (MYCMD) is also read into the pipeline because the INCL option is specified on the from-disk stage.

```
PIPE < CNMCMDU INCL DISKONLY | COLL | CONS
```

MEMLIST stage example



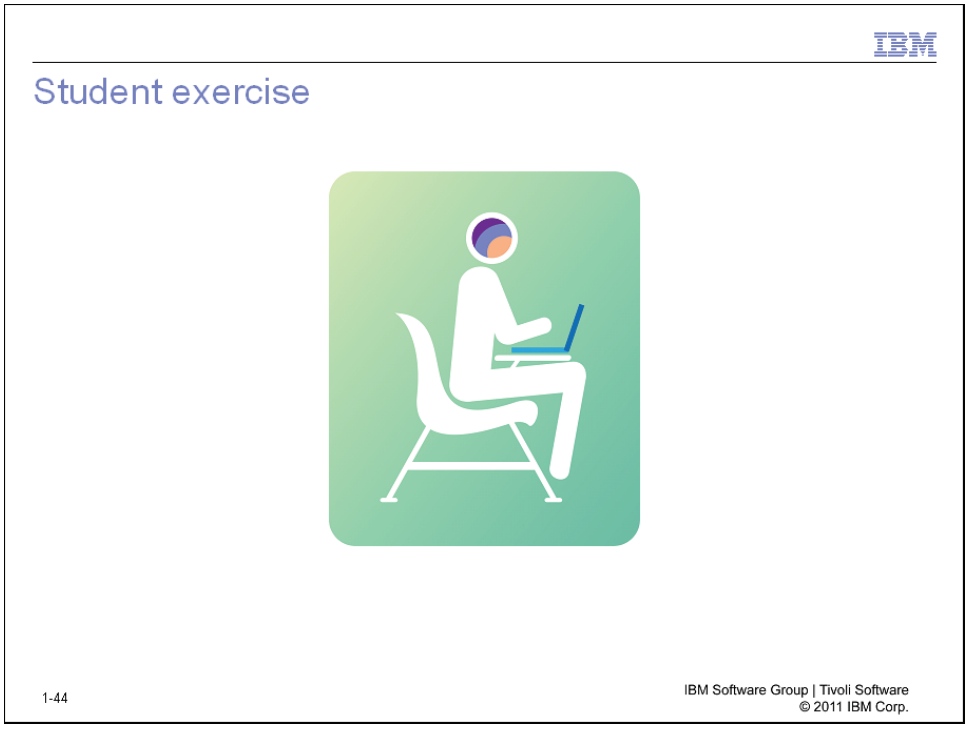
The **MEMLIST** stage lists members in NetView DD name or data set. The output is a multiline message with one line for each member as follows:

- Columns 1 to 8: Member name
- Column 10: Relative data set number
  - -1 if file is found in an operator data set
  - 0 if file is loaded in storage
  - 1 when a data set is specified

  For DD name, number matches concatenation order shown by LISTA command

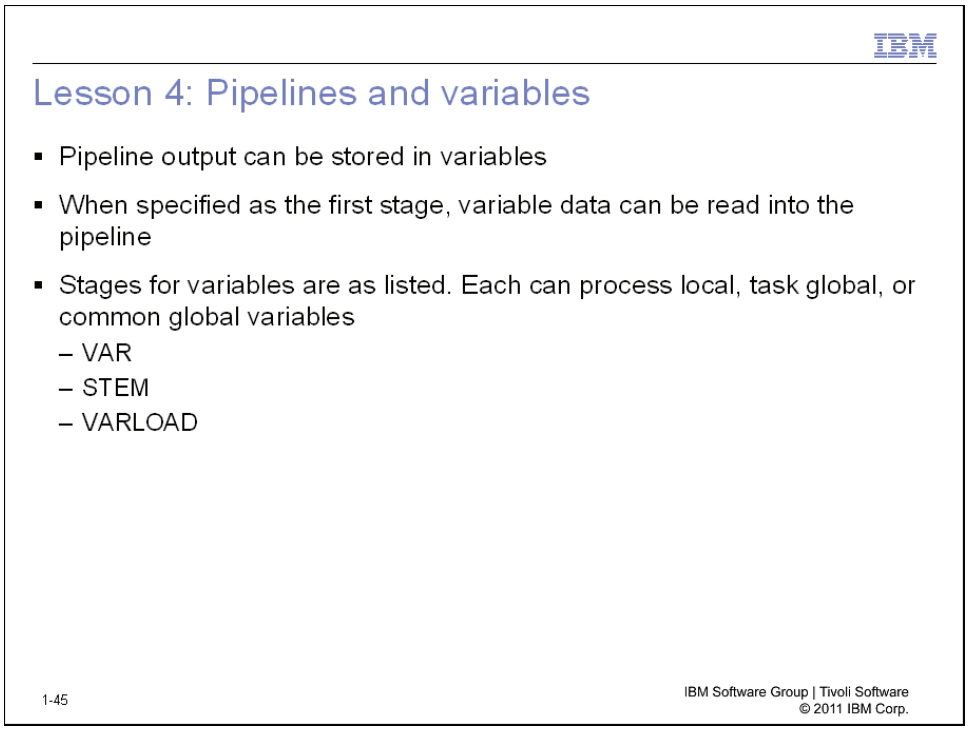
MEMLIST includes operator data sets and members that are loaded in storage by INSTORE and MEMSTORE.

Student exercise



Open your *Student Exercises* book and perform Exercise 3.

Lesson 4: Pipelines and variables



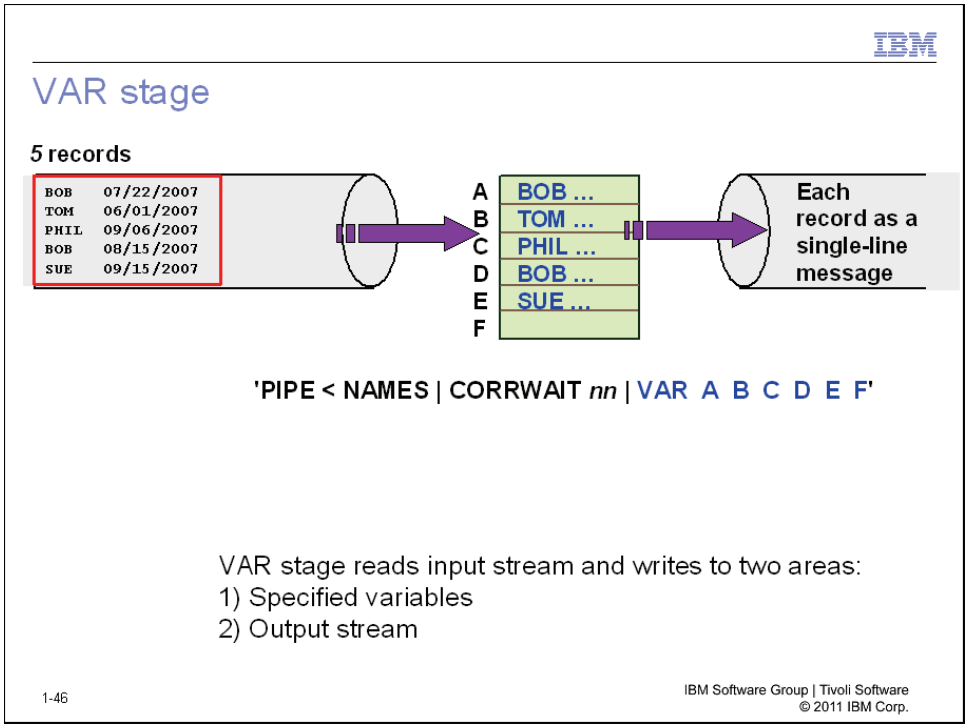
With the **VAR** stage, you can read records from or write records to variables in a command procedure variable pool. The **$VAR** stage is similar to VAR. However, $VAR also reads or writes the VIEW attribute variables (which start with $) that are associated with the specified data variables.

With the **STEM** stage, you can read or write an array of stemmed command procedure variables. The **$STEM** stage is similar to STEM. However, $STEM also reads or writes the VIEW attribute variables (which start with $) that are associated with the specified array of stemmed data variables.

With the **VARLOAD** stage, you can set values for variables that pass in the input stream. The names and values of the variables that VARLOAD sets are specified by the records that passed on the primary input stream. VARLOAD sets one variable for each input message that contains a character other than a blank or asterisk in the first position of the record.

Note: You cannot issue these stages from the NetView command line.

VAR stage



The **VAR** stage reads data from the input stream, sets the specified variables, and writes the data to the output stream. In this example, the NAMES file is read into the pipeline. The VAR stage is passed five records, setting five variables A, B, C, D, and E. Variable F is not set because only five records were passed. The VAR stage syntax is as follows:

```
.- ----.
              .- (0)------.  V       |
|--+-VAR--+--+-----------+----name-+----------------------|
   '-$VAR-'   +- (COMMON)-+
              +- (TASK)---+
              '- (number)-'
```

When VAR is the first stage of a pipeline, records are read from the variable that is specified. Each record passes as a single-line message to the pipeline output stream.

When VAR is specified as a subsequent stage, messages are read from its input stream and written to both the specified variables and to its output stream. Data from the first input message is placed

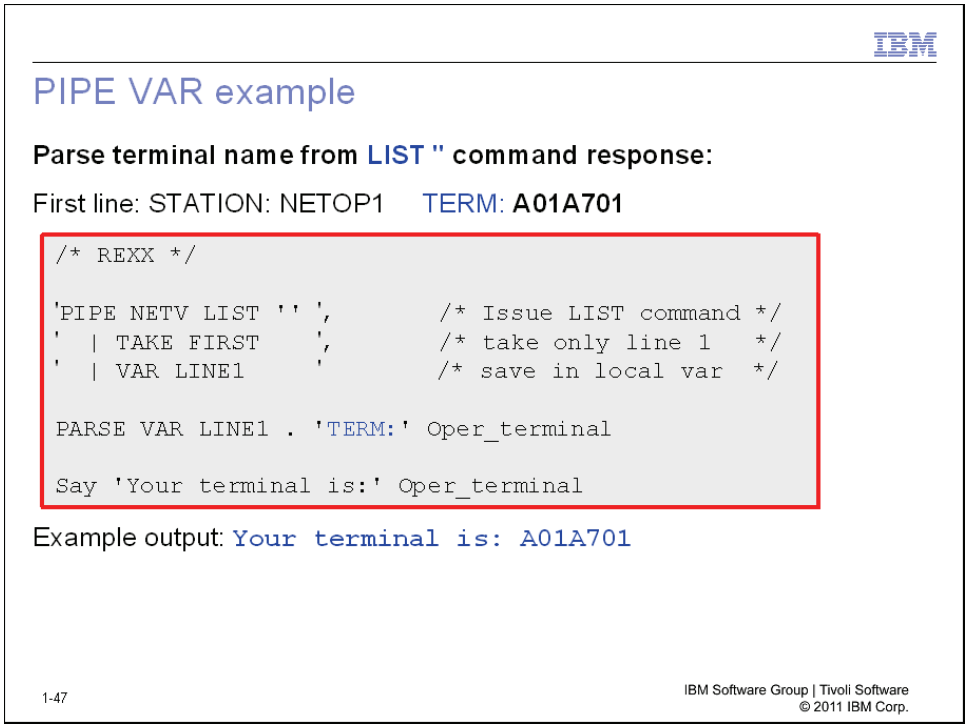
in the first variable and written to the output stream. Data from the second message goes in the second variable and to the output stream. The data flows similarly for the rest of the process.

- (COMMON): Specifies to access the common global variable dictionary instead of the local dictionary for the procedure.
- (TASK): Specifies to access the task global variable dictionary instead of the local variable dictionary for the procedure.
- (*number*): Specifies the number of invocations (generations) to refer back when setting the variables. The default is zero (0).
- *name* - Specifies the name of the variable to read from or write to.

Tip: Pipeline variables are dropped before being used. This can cause errors if you use SIGNAL ON NOVALUE conditions in REXX programs. Use REXX SYMBOL() function to test for null variables: An example follows:

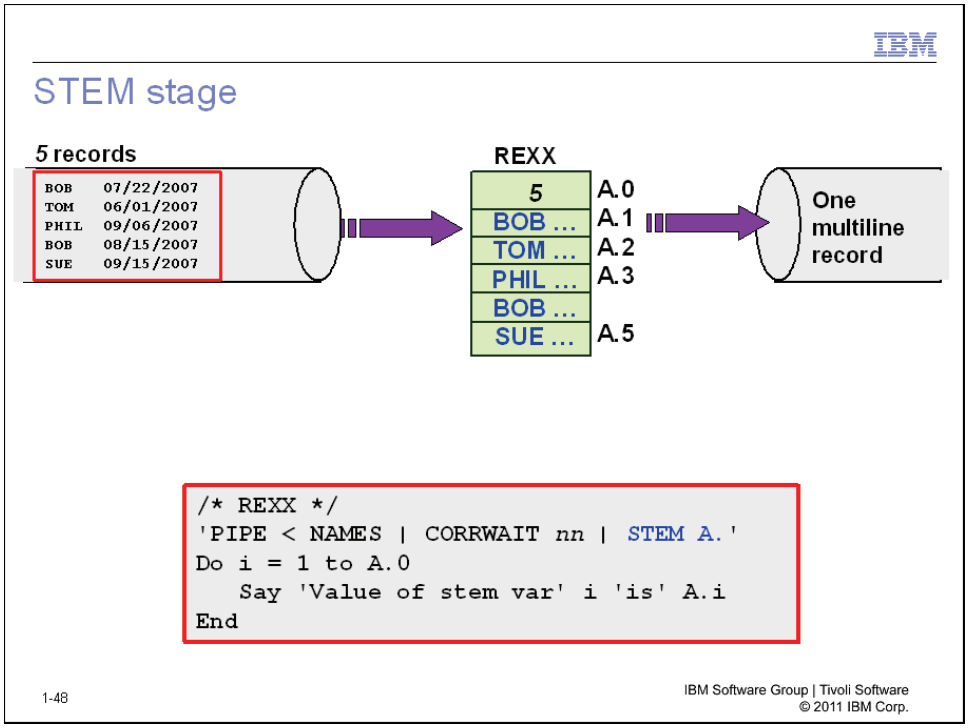
```
IF Symbol(E) <> 'LIT' then   /* If variable E is not null */
```

PIPE VAR example



In this example, the NetView **LIST** '' command is issued. Several single-line messages are generated. The pipeline takes the first message and places it into the LINE1 variable without stopping at the other messages. The next instruction, REXX **PARSE VAR**, parses LINE1 for the operator terminal.

STEM stage

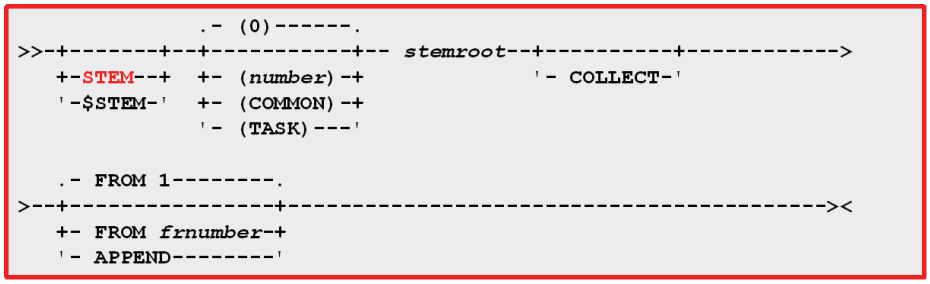


The **STEM** stage reads data from the input stream, sets the specified stem variables, and writes the data to the output stream. In this example, the **STEM** output would be as follows:

```
Value of stem var 1 is BOB 07/22/2007
Value of stem var 2 is TOM 06/01/2007
Value of stem var 3 is PHIL 09/06/2007
Value of stem var 4 is BOB 08/15/2007
Value of stem var 5 is SUE 09/15/2007
```

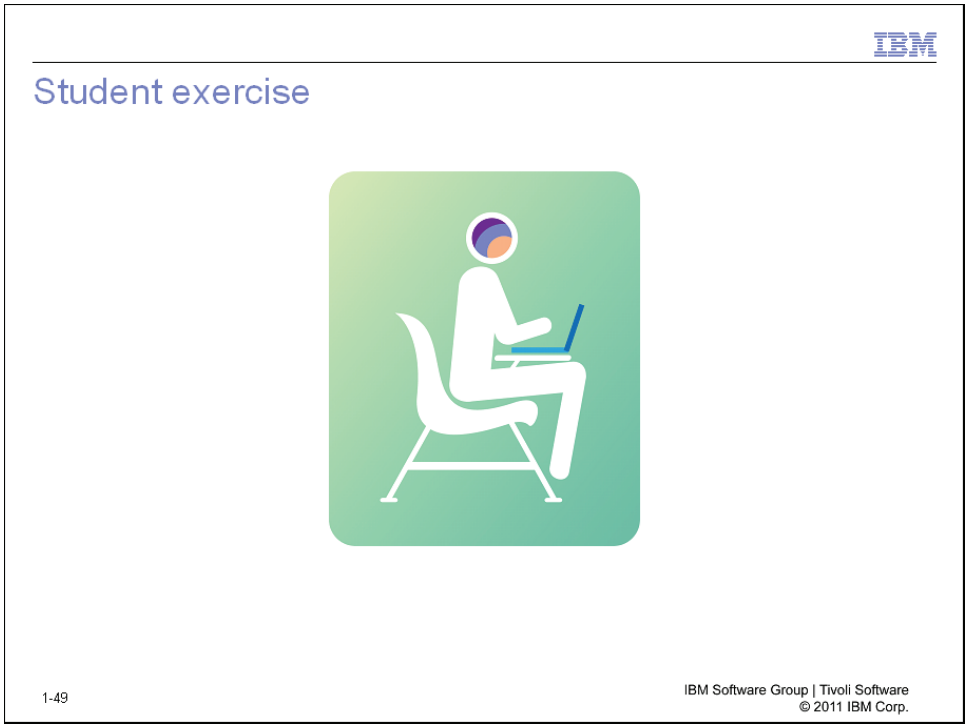
- When STEM is the first stage, it reads records from an array of stemmed variables. Each record passes as a single-line message to the pipeline output stream.
- When STEM is used for reading records into a stem variable, the count of stem variables automatically updates in element zero.
- When STEM is not the first stage, it writes each line of each a message to stem variables and to the output stream. In addition, an integer appends to the given variable name (for example, VARNAME*n*).
- When STEM is used for writing records to the pipeline, the number of records to write must be defined in element zero.

The STEM stage syntax is as follows:



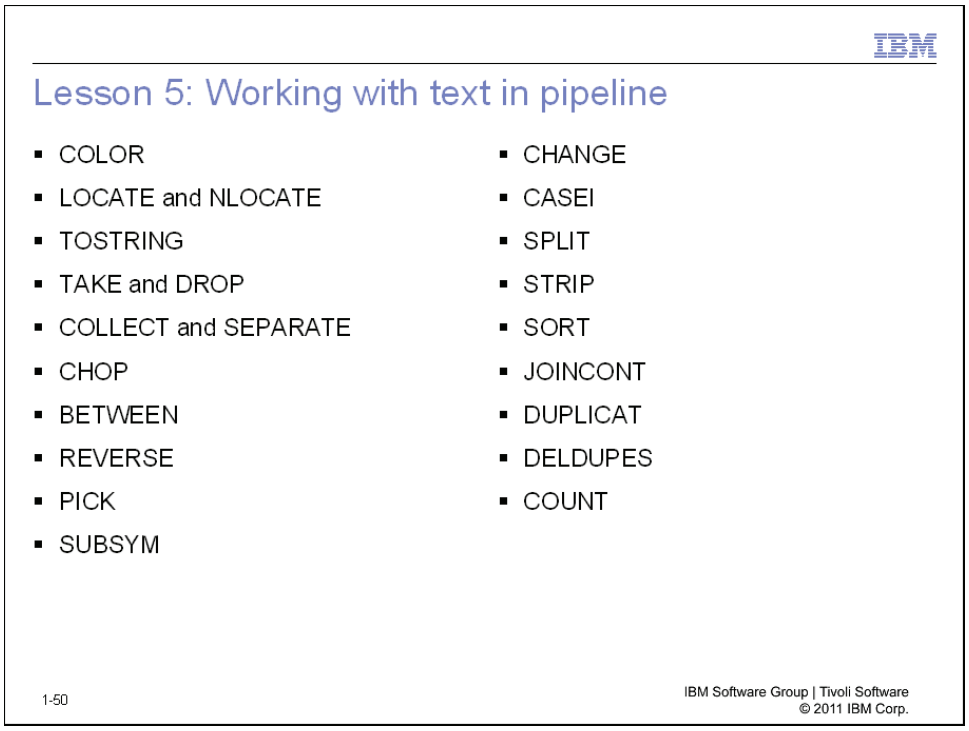
- *stemroot*: Specifies the name of the stem variable to read from or write to. It should end with a period (.) if you use a REXX EXEC.
- APPEND: Specifies that new data should be appended as additional stem variables at the end of the current stem variables. APPEND can be used only on a stage that is not first.
- COLLECT: Causes STEM to build one multiline message instead of many single-line messages. COLLECT is allowed only when STEM is the first stage in a pipeline.
- FROM: Indicates a starting point for access to the stem variables.

Student exercise



Open your *Student Exercises* book and perform Exercise 4.

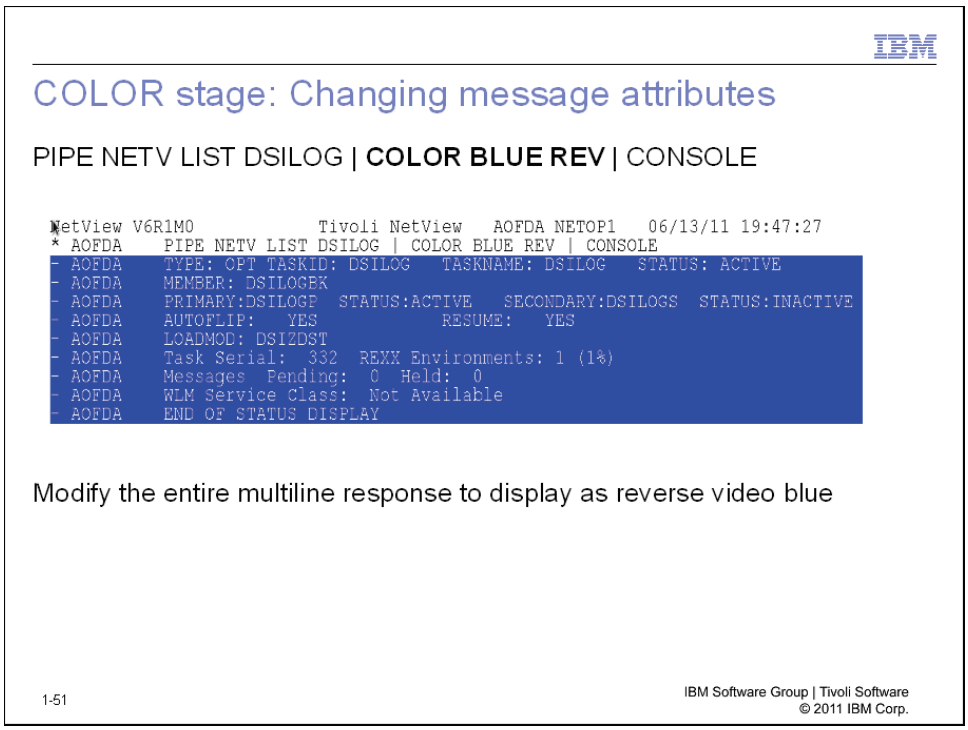
Lesson 5: Working with text in pipeline



This lesson contains information on several pipeline stages for modifying the contents of pipeline data. Use the NetView HELP PIPE STAGES command to display a panel with a list of all NetView pipe stages. On the panel, **tab over** to a stage name (for example, LOCATE) and press Enter to view the help for the LOCATE stage.

Manipulated pipeline data routes to the primary output stream of these pipe stages with all other data sent to the secondary output stream.

COLOR Stage: Change message attributes

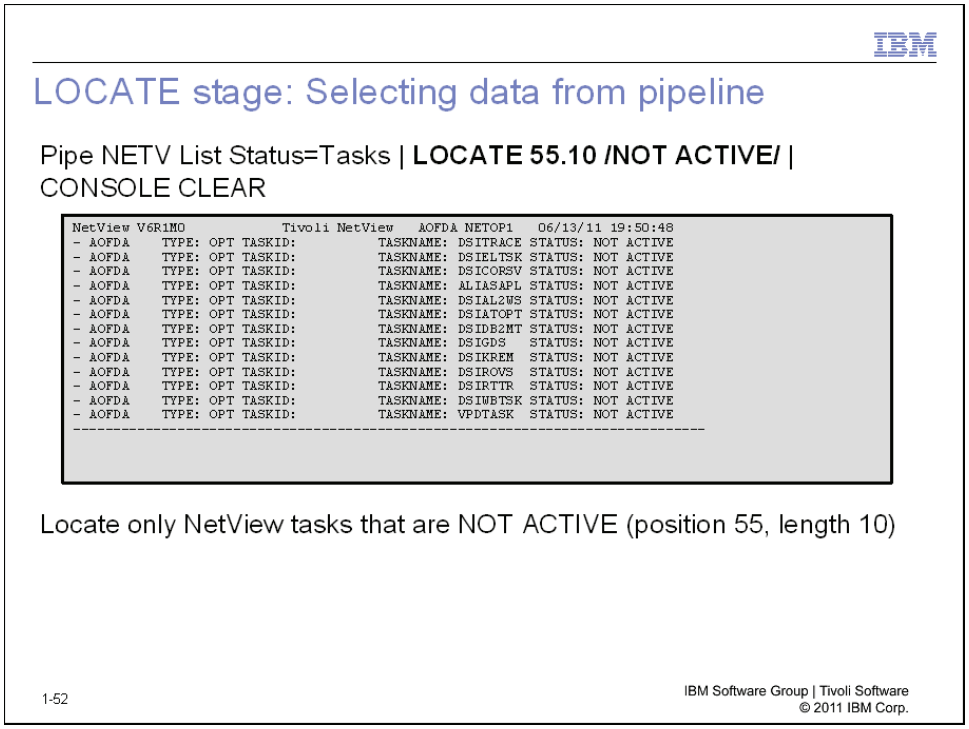


The **COLOR** (presentation attributes) stage changes how messages look at the NetView console. You can change the color, highlighting, and intensity of data records in a pipeline:

- Color: Blue, red, pink, yellow, green, white, turquois, default
- Highlighting: Reverse, undersco, blink, none
- Intensity: Normal, bright, dark

Synonyms of COLOR are COLOUR and PRESATTR.

LOCATE stage: Selecting data from pipeline



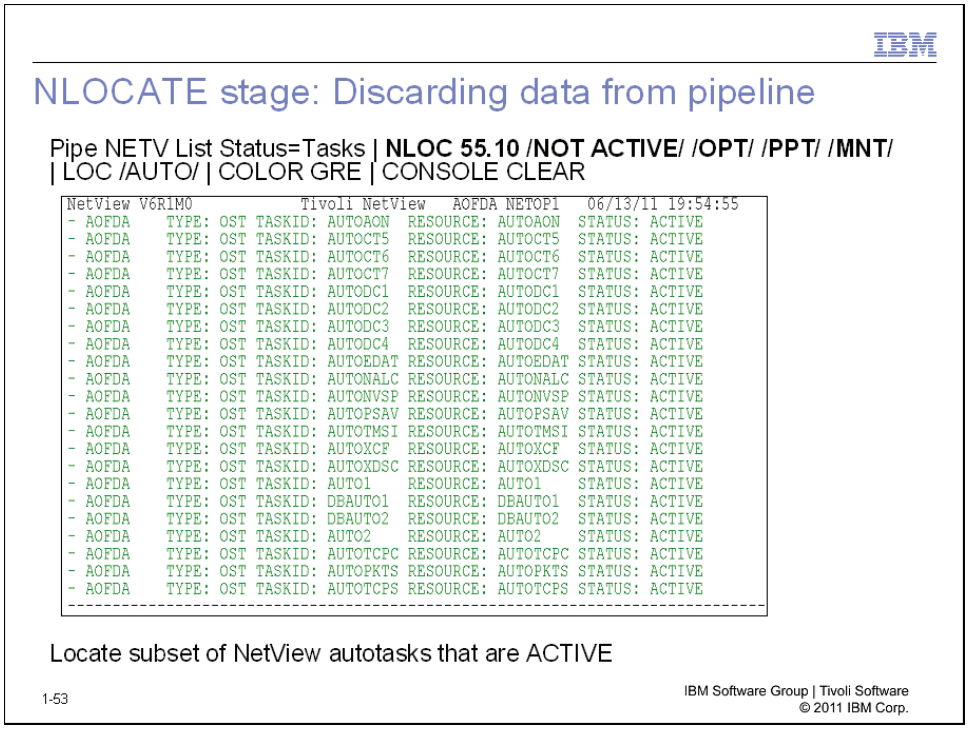
The **LOCATE** stage selects messages that match a specified delimited character string to be passed to the primary output stream. Messages that do not contain the character string pass to the secondary output stream, if connected. If no secondary output stream exists, the messages that do not match the LOCATE are discarded.

LOCATE examines each input message for a match to the delimited string. You can supply a position and length pair to limit the search to a particular column range. An example follows:

```
'Pipe NETV List Status=Tasks',
      '|LOCATE 55.10 /NOT ACTIVE/ | CONSOLE CLEAR'
```

This stage issues the NetView **LIST STATUS=TASKS** command and places the output in the pipeline. Only records that contain the string NOT ACTIVE beginning in position 55 for a length of 10 are selected.

NLOCATE stage: Discarding data from pipeline



The **NLOCATE** stage discards messages from the primary output stream that match a specified delimited character string. Messages that do not contain the character string are passed to the primary output stream. Messages that contain the character string are passed to the secondary output stream, if connected.

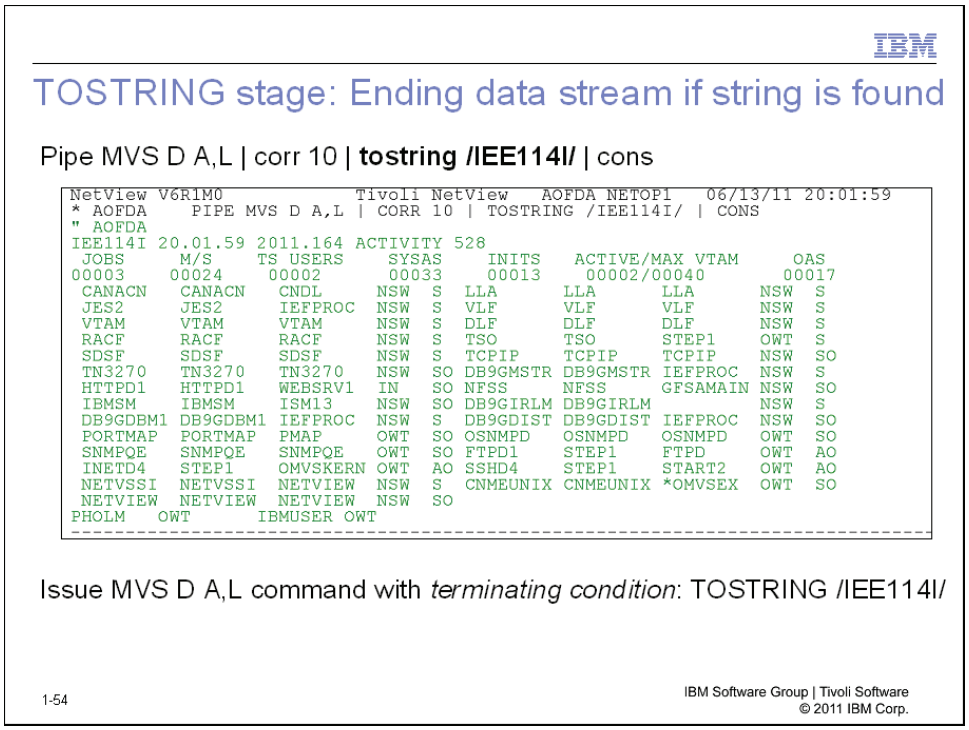
You can supply a position and length to limit the search to a particular column range. An example follows:

```
'Pipe NETV List Status=Tasks',
      '|NLOC 55.10 /NOT ACTIVE/ | COLOR GRE |CONSOLE CLEAR'
```

This stage issues the NetView **LIST STATUS=TASKS** command and places the output in the pipeline. Only the records that do not contain the string NOT ACTIVE beginning in position 55 for a length of 10 are selected.

The example on the slide contains other data to NLOCATE. In this case, data for all tasks of type OPT, PPT, and MNT are to be discarded. The LOCATE stage further refines the pipeline data by finding tasks that begin with AUTO.

TOSTRING stage: Ending data stream if string is found

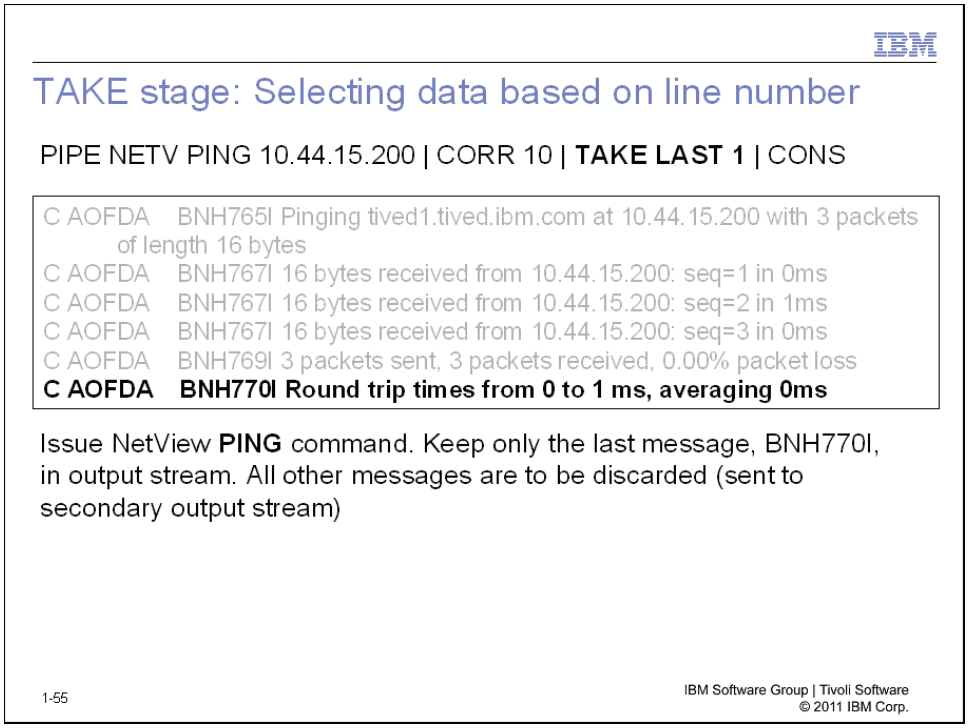


With the **TOSTRING** stage, you can select messages containing the text that match a specified string. Selected messages pass to the primary output stream. Those not selected pass to the secondary output stream, if connected. TOSTRING generates a *terminating condition* for the CORRWAIT stage. TOSTRING ends the preceding CORRWAIT stage.

In the example, if you do not specify a terminating condition, the IEE114I is displayed, but the pipeline waits the full 10 seconds before it disconnects and moves to the CONS stage.

You can specify up to 40 delimited strings, each with an optional position and length pair to limit the column range of the search.

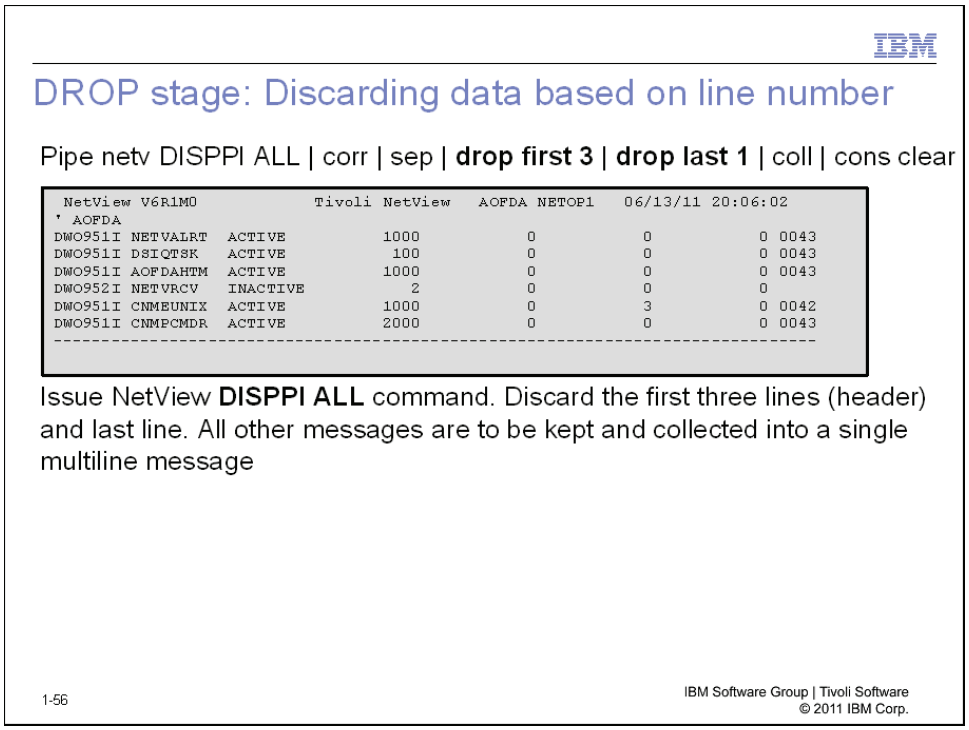
TAKE stage: Selecting data based on line number



With the **TAKE** stage, you can specify the number of messages or lines that pass to the primary output stream (if any). All messages or lines that exceed this number pass to the secondary output stream.

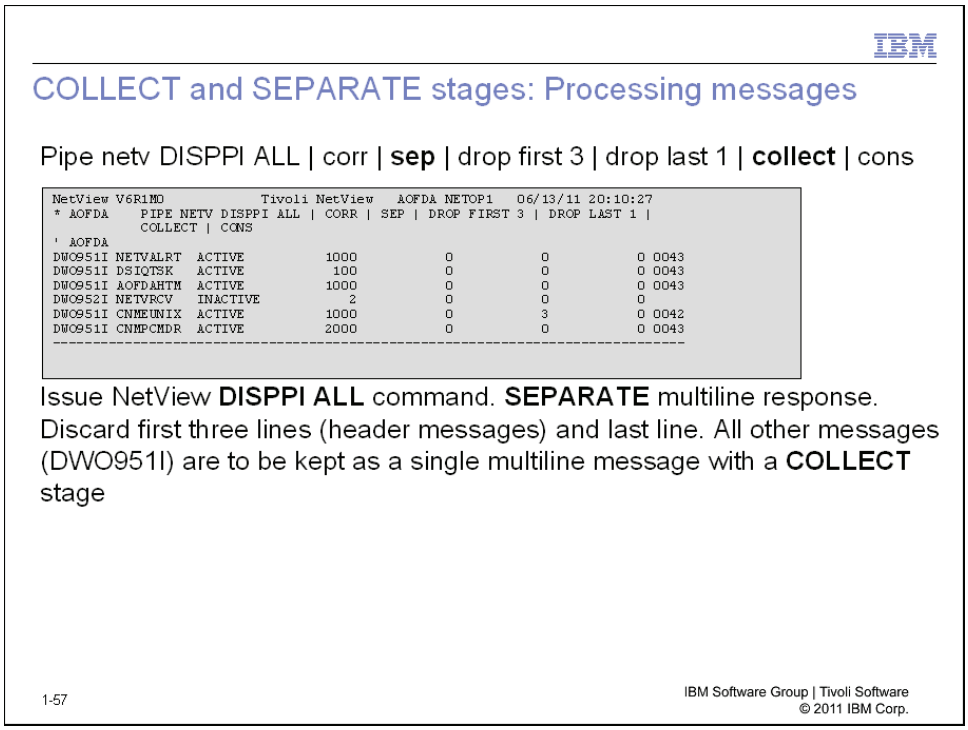
TAKE disconnects the input stream with discarded messages that pass to the secondary output stream. TAKE generates a *terminating condition* for the CORRWAIT stage. TAKE LAST *nn* waits until the last message is displayed (or a timeout occurs).

DROP stage: Discarding data based on line number



With the **DROP** stage, you can specify how many messages or lines are to be discarded from the primary output stream. When the specified number of messages are read from the input stream and discarded, all other messages copy to the primary output stream. Discarded messages or lines pass to the secondary output stream, if connected.

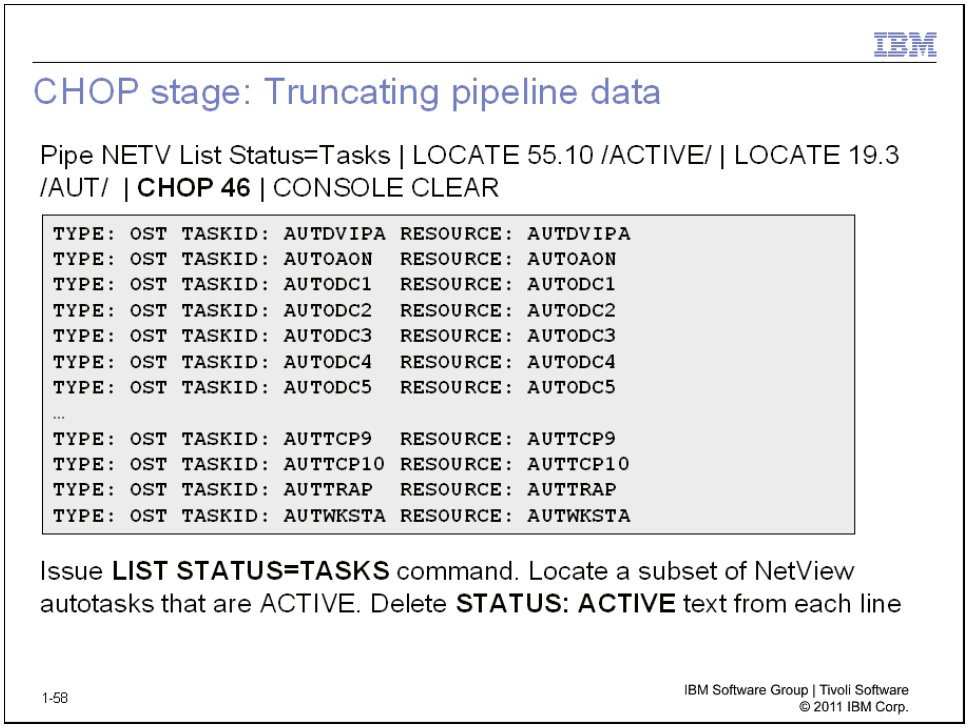
COLLECT and SEPARATE stages: Processing messages



The **SEPARATE** stage transforms multiline messages into multiple single-line messages. Input single-line messages pass without modification. Output single-line messages inherit all of the attributes of the input messages that created them. The output of SEPARATE consists of single-line messages.

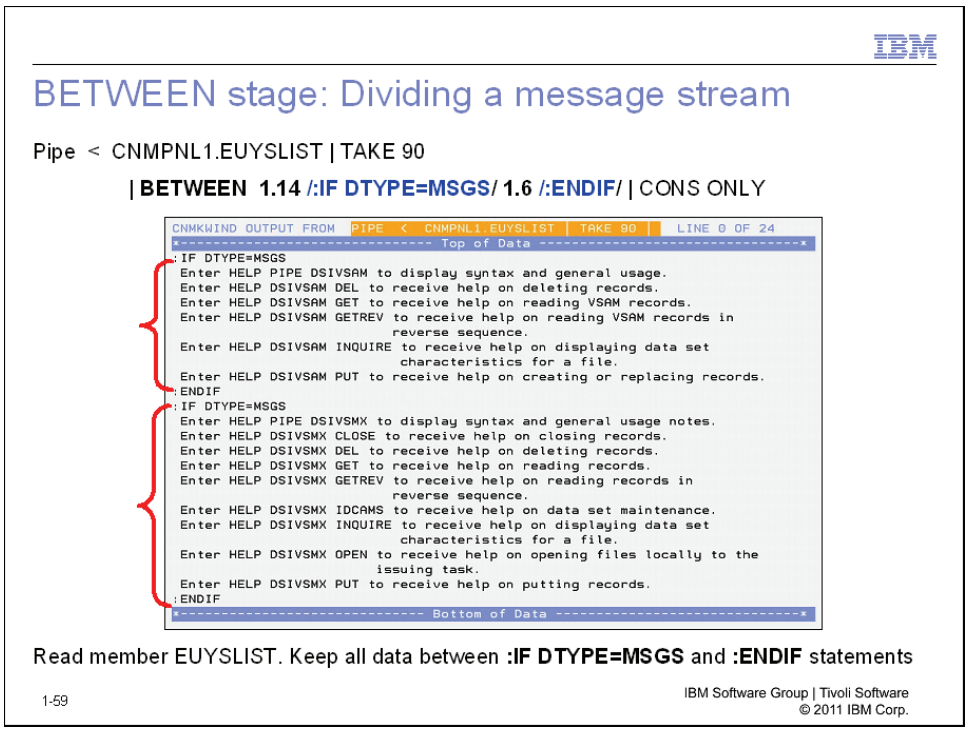
The **COLLECT** stage creates a multiline message from one or more messages in the pipeline. The attributes of the output are inherited from the first line that is collected for that message.

CHOP stage: Truncating pipeline data



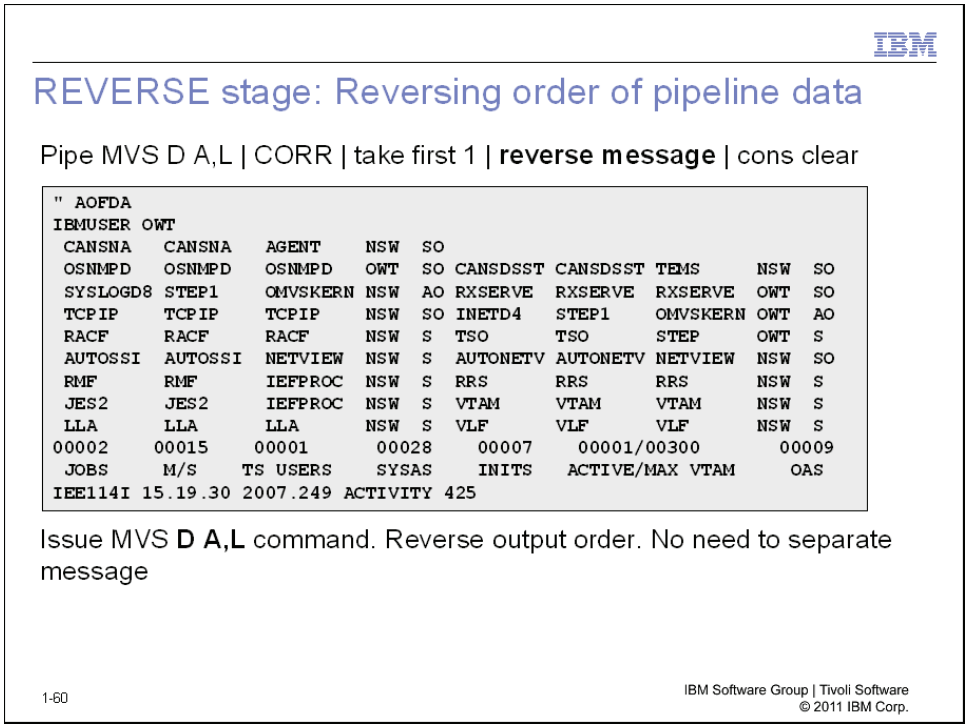
The **CHOP** stage truncates lines after a specified column, character, or string. The data that CHOP keeps passes to the primary output stream. The data that CHOP discards passes to the secondary output stream, if connected. In this example, all data after column 46 is deleted. This action removes STATUS: ACTIVE from each line in the pipeline.

BETWEEN stage: Dividing a message stream



The **BETWEEN** stage divides a message stream into sections. The selected sections begin with a message containing a specified string and end with either a specified string or a number of messages. The selected sections pass to the primary output stream. The sections that were not selected pass to the secondary output stream. All lines in EUYSLIST between the :IF DTYPE=MSGS and :ENDIF statements are included in the pipeline data.

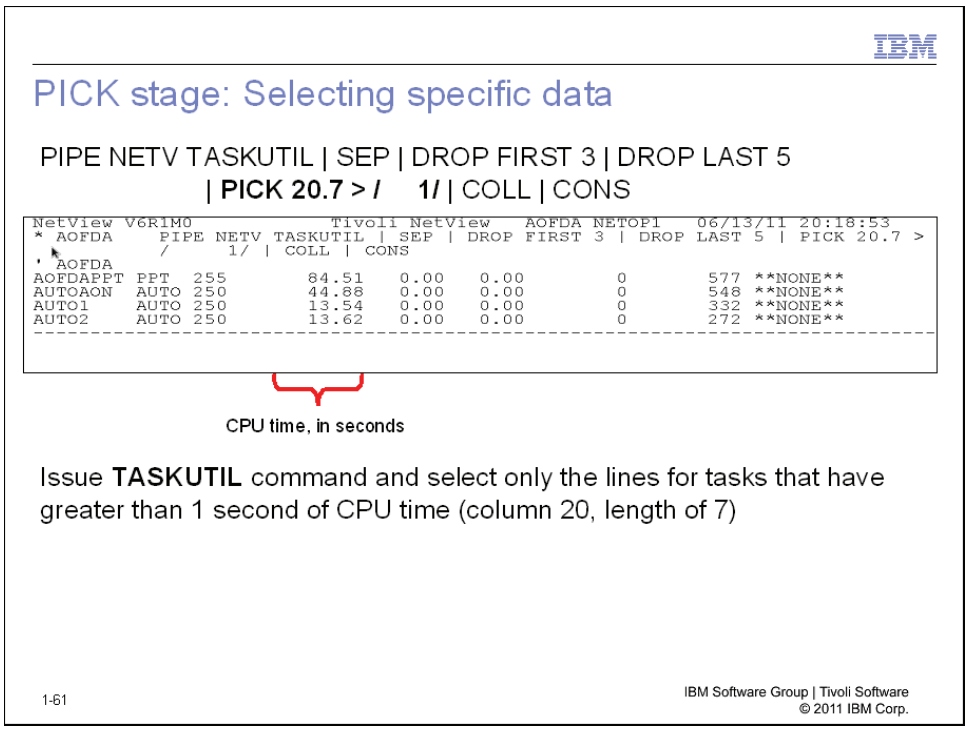
REVERSE stage: Reversing order of pipeline data



You can use **REVERSE** stage to reverse the order of data in the pipeline as follows:.

- LINE (default): Specifies that each line of output is to be reversed, character by character. For example, DSI069I SYNTAX ERROR changes into RORRE XATNYS I960ISD.
- MESSAGE: Specifies that each multiline message is to be reversed, line by line. Affects only multiline messages.
- STREAM: Specifies that the next stage receives messages in reverse order. This option has no effect on the structure of multiline messages.
- REVERSE waits until all messages are in the pipeline before processing the data.

PICK stage: Selecting specific data

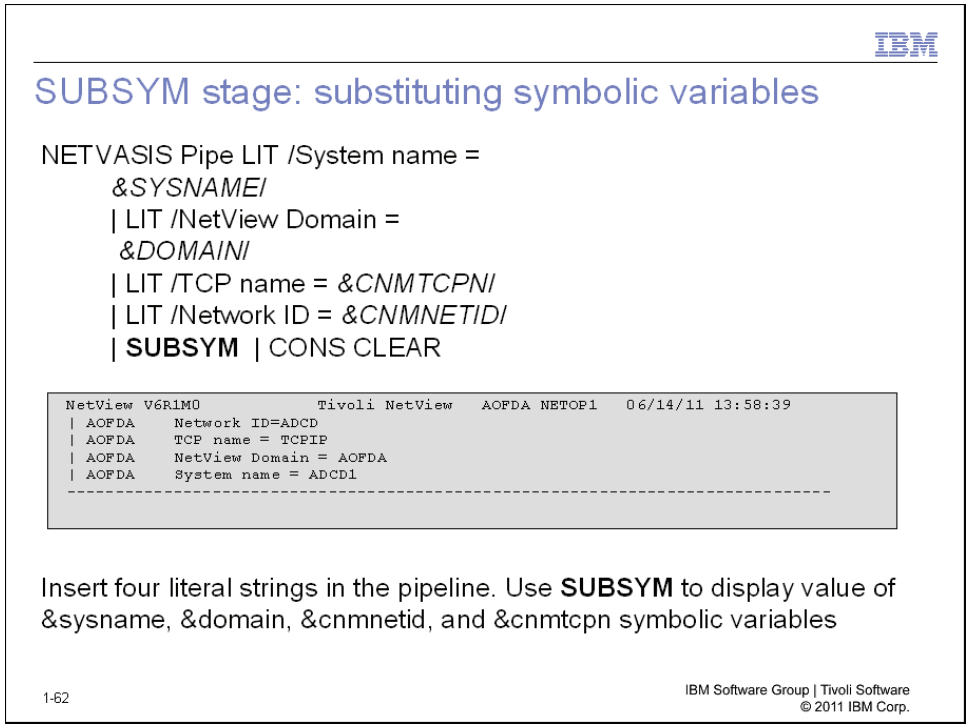


The **PICK** stage selects messages satisfying a criteria and passes them to the primary output stream. Messages that do not meet the specified criteria pass to the secondary output stream, if connected.

The **PICK** stage examines only the first line of a multiline message. If selected, the entire multiline message passes to the primary output stream. Use the SEParate stage to break a multiline message into multiple single-line messages.

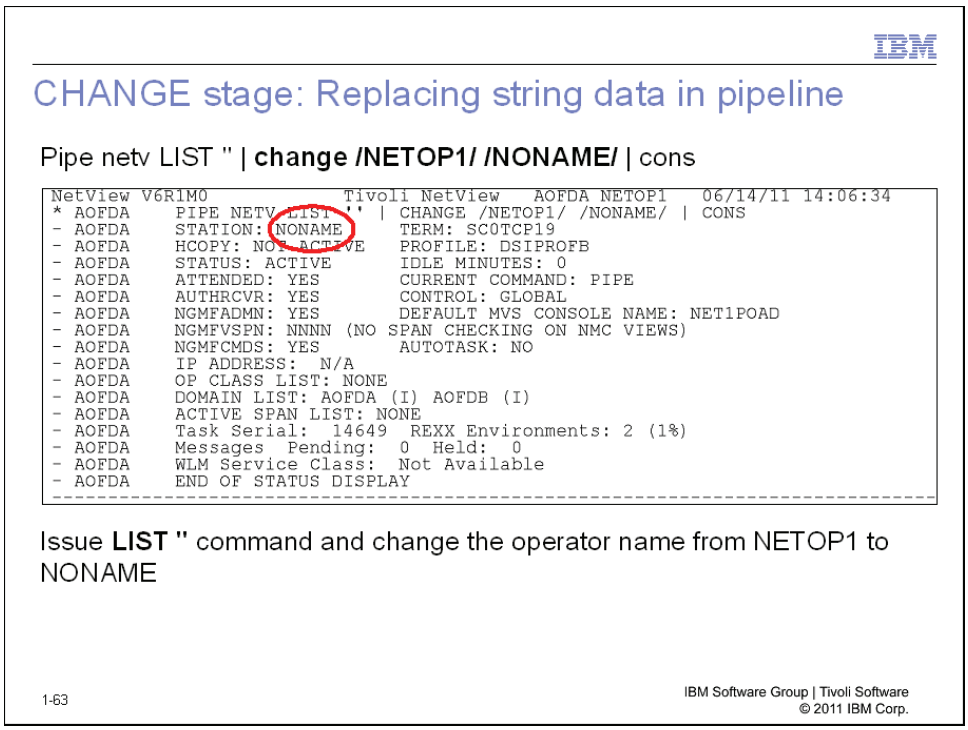
Specify the selection criteria by giving a *position.length* within the message line to be compared against: another *position.length* within the message line or a character string. In this example, the TASKUTIL output is interrogated for any task that has consumed more than 1 second of CPU time. Because the field being checked has a length of seven, the string that is specified on PICK must contain seven characters between the delimiters. The delimiters are forward slashes.

SUBSYM stage: Substituting symbolic variables



The **SUBSYM** stage substitutes system or user-defined symbolic variables (including NetView &DOMAIN) into messages in the pipeline. In this case, NETVASIS is used for retaining the mixed case of the literal strings also. To preserve the order of the strings, you can insert a REVERSE STREAM before the CONSOLE stage.

CHANGE stage: Replacing string data in pipeline

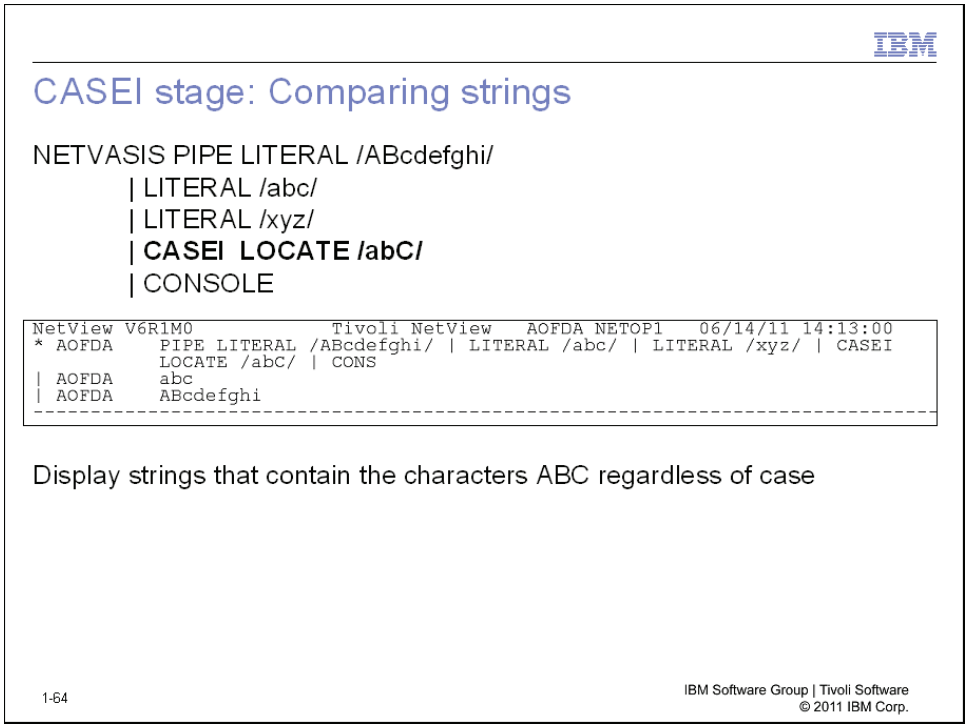


The **CHANGE** stage replaces occurrences of the first specified string with the second string. Either string can be null. If the first string is null, the second string inserts at the beginning of each line. If the second string is null, all occurrences of the first one are deleted. Data between substitutions are copied without change.

If a secondary output stream is defined, changed messages pass to the primary output stream. Unchanged messages (because first string is not found) pass to the secondary output stream.

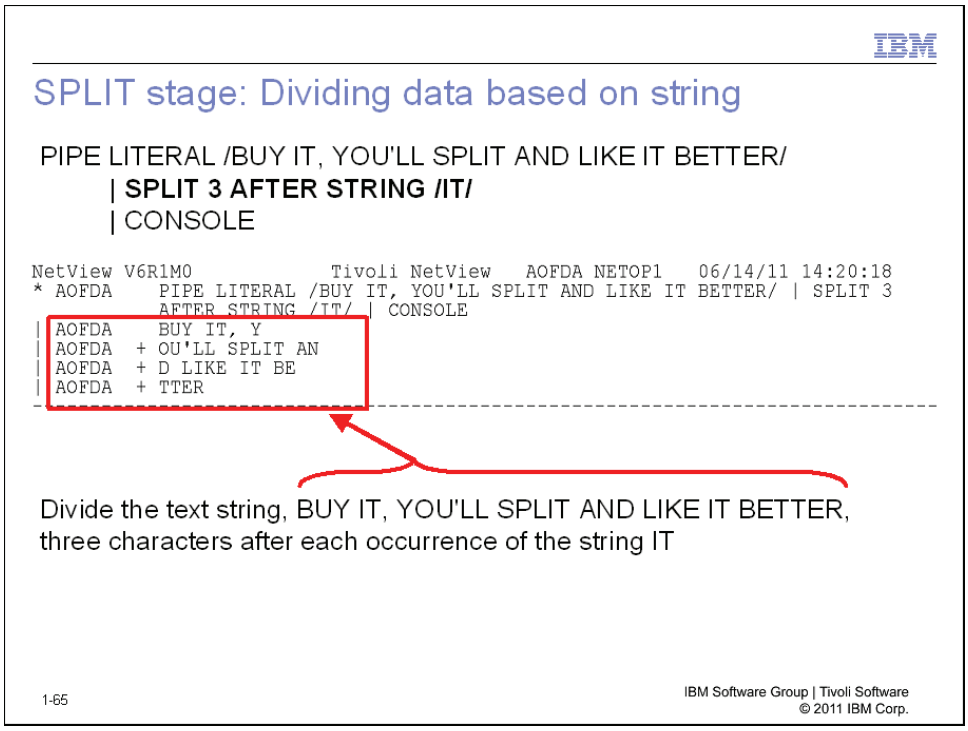
In this example, operator NETOP1 issues the LIST '' command. Notice the STATION: NONAME in the first line of the response.

CASEI stage: comparing strings



The **CASEI** stage compares character strings without case sensitivity to uppercase or lowercase EBCDIC characters. In this example, CASEI LOCATE /abC/ matches *ABcdefghi* and *abc*.

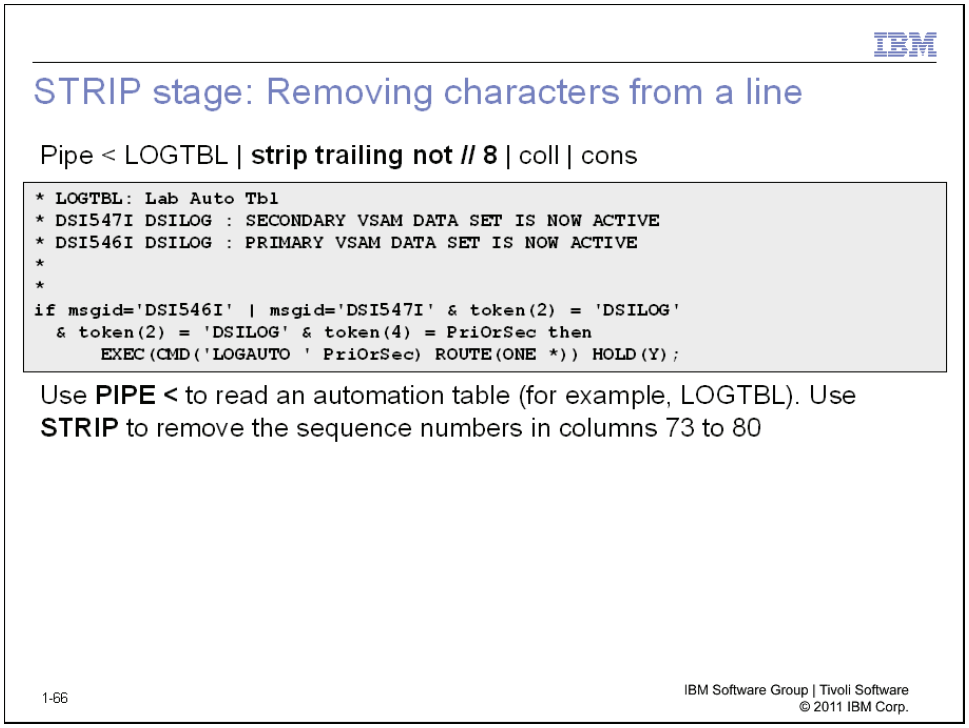
SPLIT stage: Dividing data based on string



The **SPLIT** stage divides a line of text into multiple lines, based on a specified string. The default delimiter is a blank. The **SPLIT 3 AFTER STRING /IT/** example displays the following text:

```
BUY IT, Y
OU'LL SPLIT AN
D LIKE IT BE
TTER
```

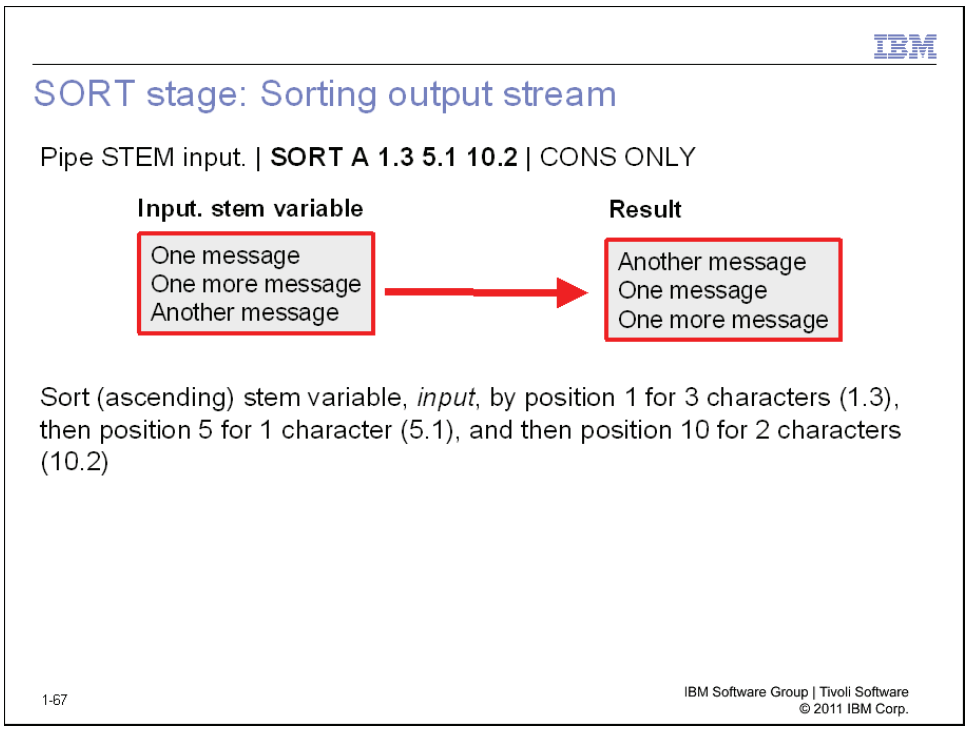
STRIP stage: Removing characters from a line



The **STRIP** stage removes blanks or other specified characters from the beginning or end of message data. Alternately, STRIP removes all characters up to a blank or other specified characters.

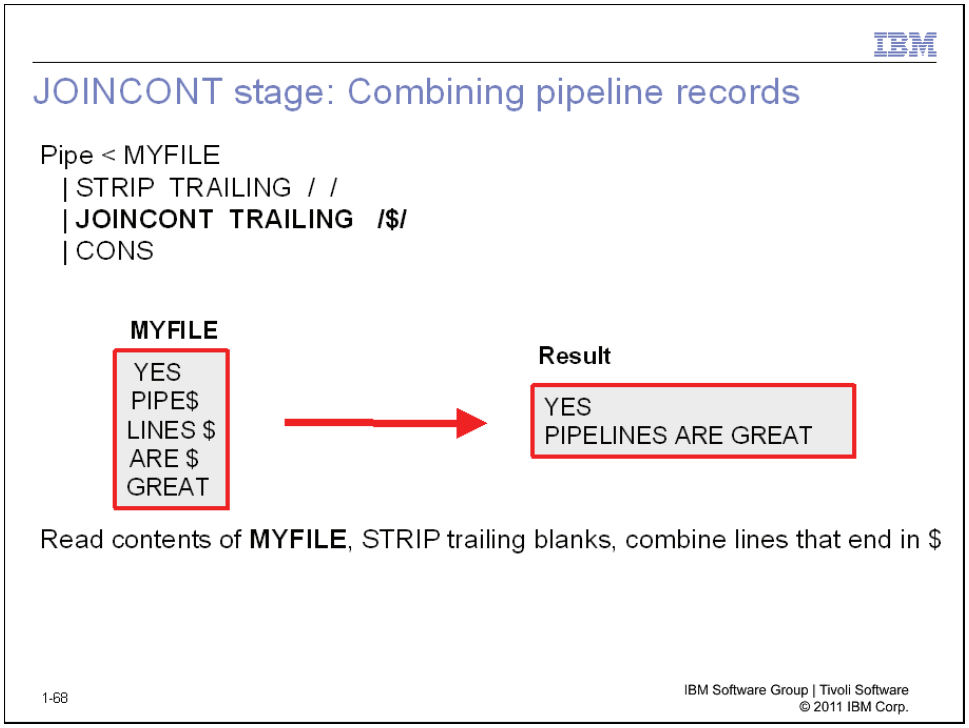
Tip: STRIP can remove unwanted blanks or other characters before you use the JOINCONT stage. Discussion about JOINCONT occurs later.

SORT stage: Sorting output stream



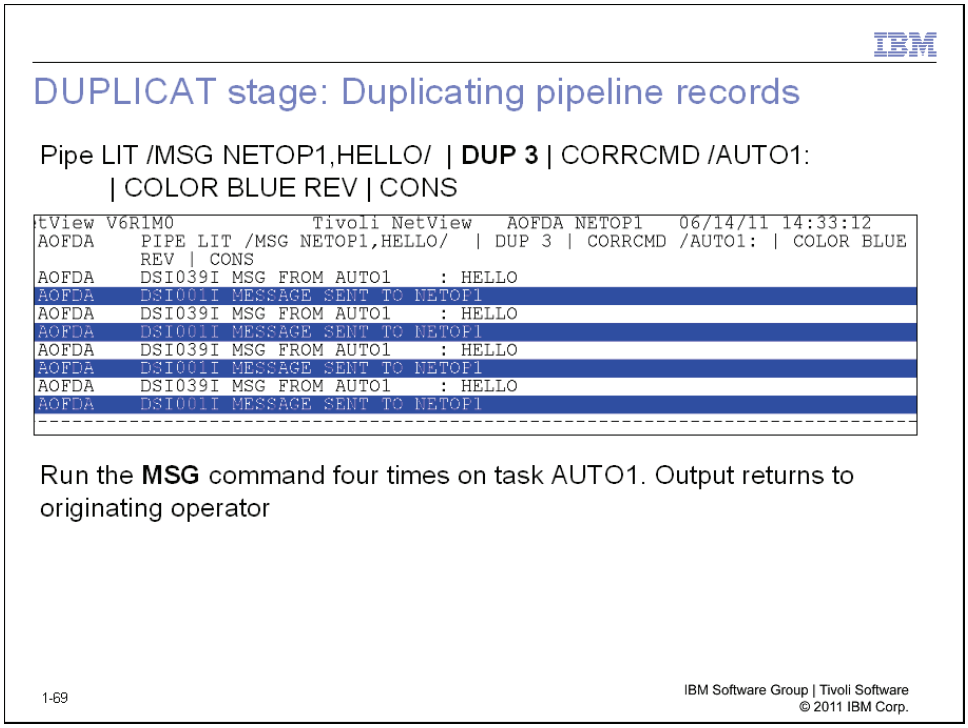
The **SORT** stage reads messages from the input stream and writes them to the output stream in a specified order, ascending or descending. Only the first line of each multiline message is examined. To sort lines within a multiline message, the SEPARATE stage must be included before SORT. If messages contain identical sort fields, they retain their input stream order when passed to the output stream.

JOINCONT stage: Combining pipeline records



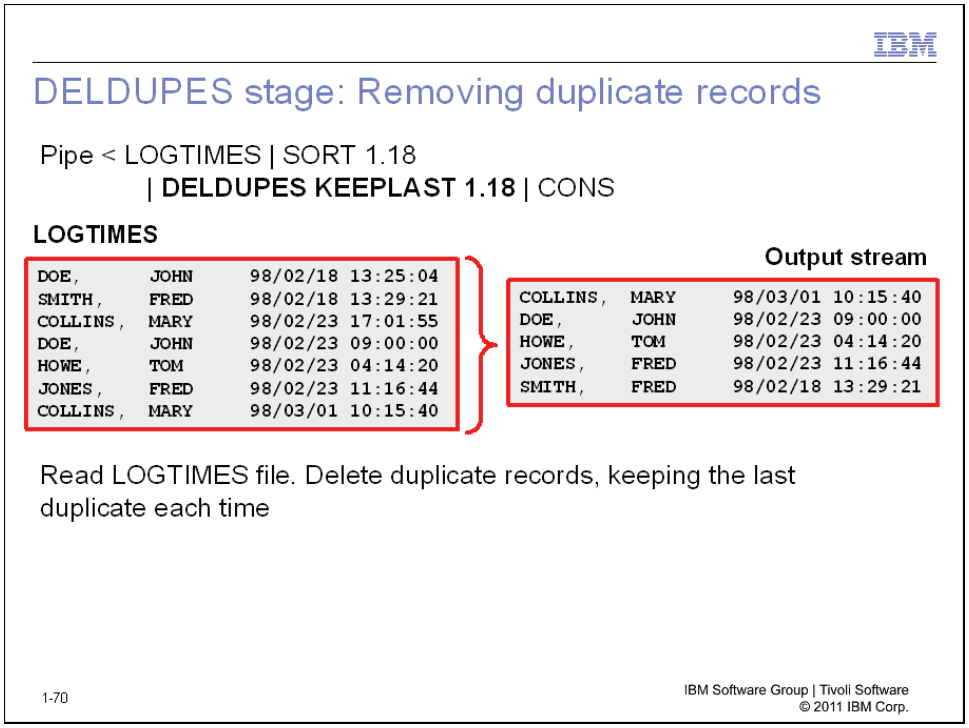
The **JOINCONT** stage joins consecutive messages in the pipeline when a match to a specified string is found. A message is considered in its entirety, and it can include blanks or even sequence numbers if reading from a file. In this example, the pipeline records that contain a $ are joined. MYFILE contains five lines. The result contains two lines.

DUPLICAT stage: Duplicating pipeline records



The **DUPLICAT** stage takes the messages in the input stream, copies them, and writes the copied messages to the output stream. The copies are marked as *copy* rather than *primary*. The message text does not change. In this example, the output from the pipeline is the DSI001I message. The DSI039I message results from the MSG NETOP1,HELLO command.

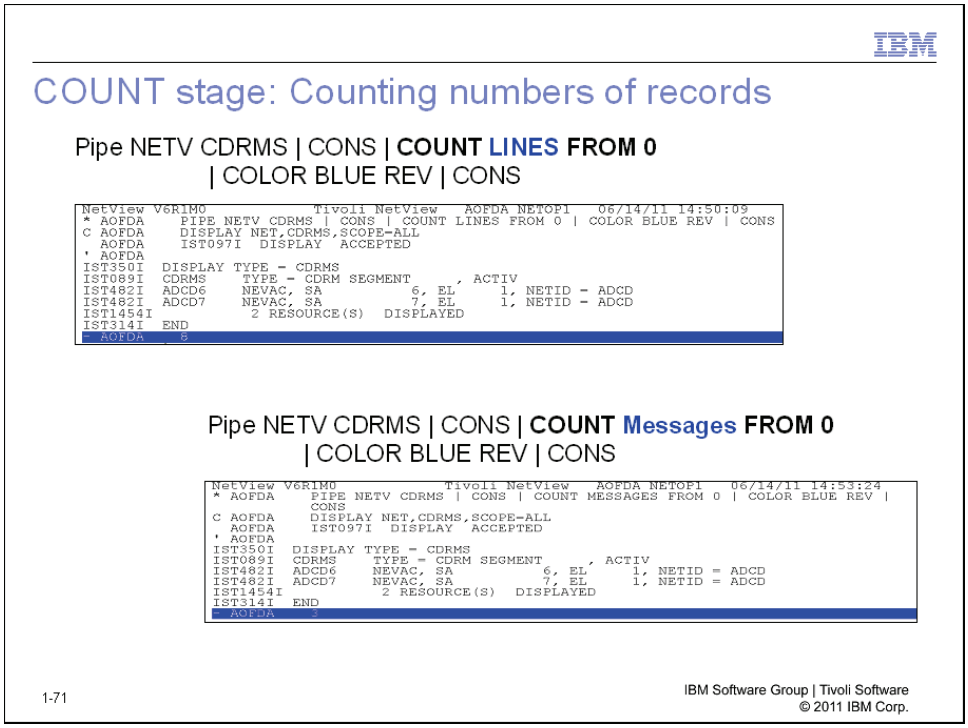
DELDUPES stage: Removing duplicate records



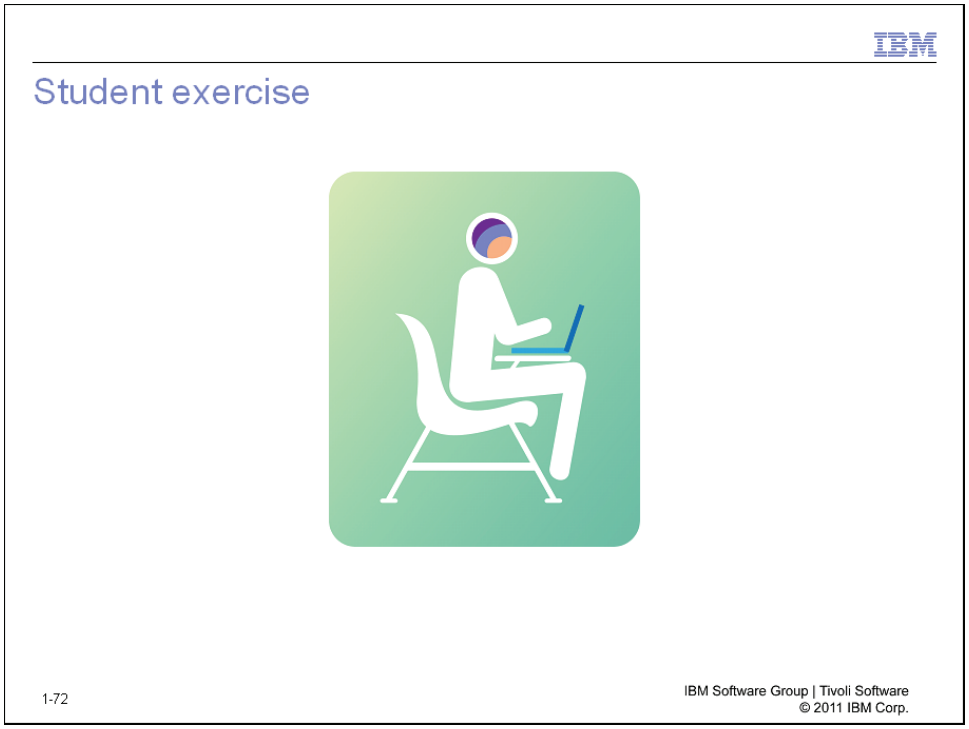
The **DELDUPES** stage compares the first line of consecutive messages and deletes consecutive duplicates. The duplicate messages are written to the secondary output stream, if connected. To delete duplicate lines within a multiline message, use the SEPARATE stage before the DELDUPES stage.

In this example, duplicate records are removed from the LOGTIMES file based on columns 1 to 18 (last name). The first and third records are removed from the output stream. The LOGTIMES file does not change.

COUNT stage: Counting numbers of records

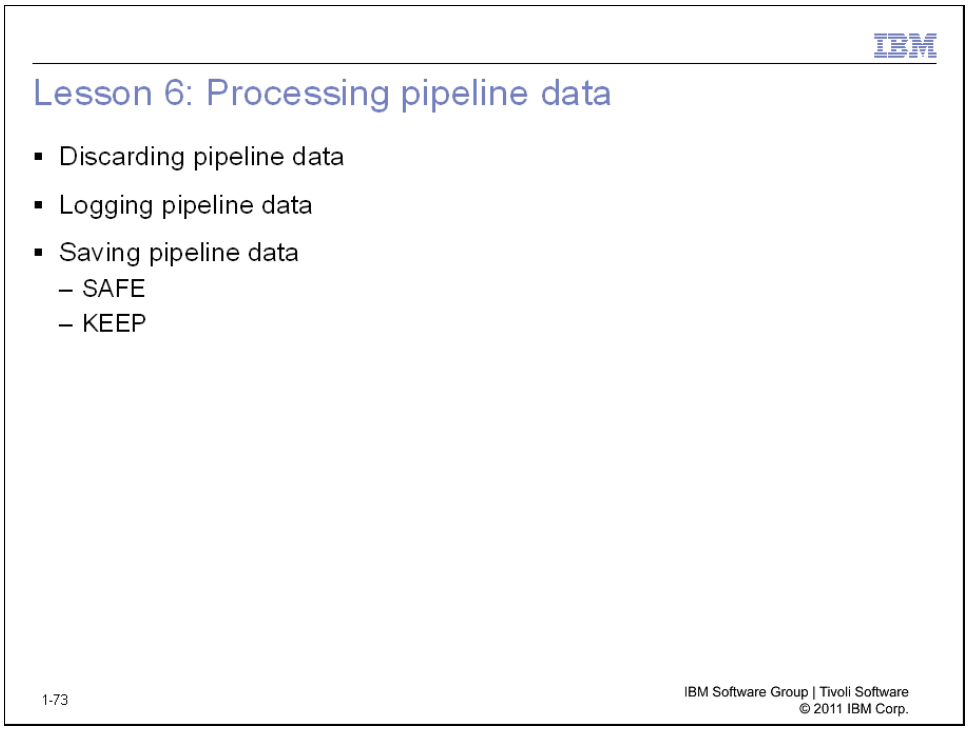


The **COUNT** stage counts the number of messages, lines, or bytes received on its primary input stream. The count passes to its primary output stream when the input stream disconnects. The first example counts the number of lines on the pipeline (eight). The second example counts the number of messages in the pipeline (three): the DISPLAY command, IST097I, and IST350I. The first CONS stage is used for these examples to show the contents of the pipeline.



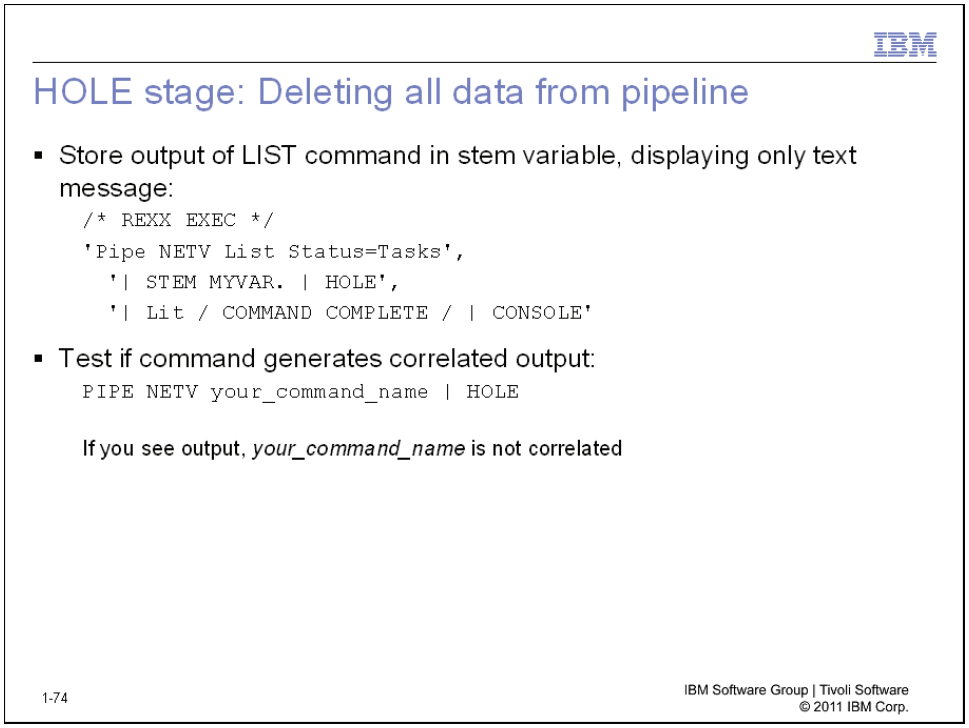
Open your *Student Exercises* book and perform Exercise 5.

Lesson 6: Processing pipeline data



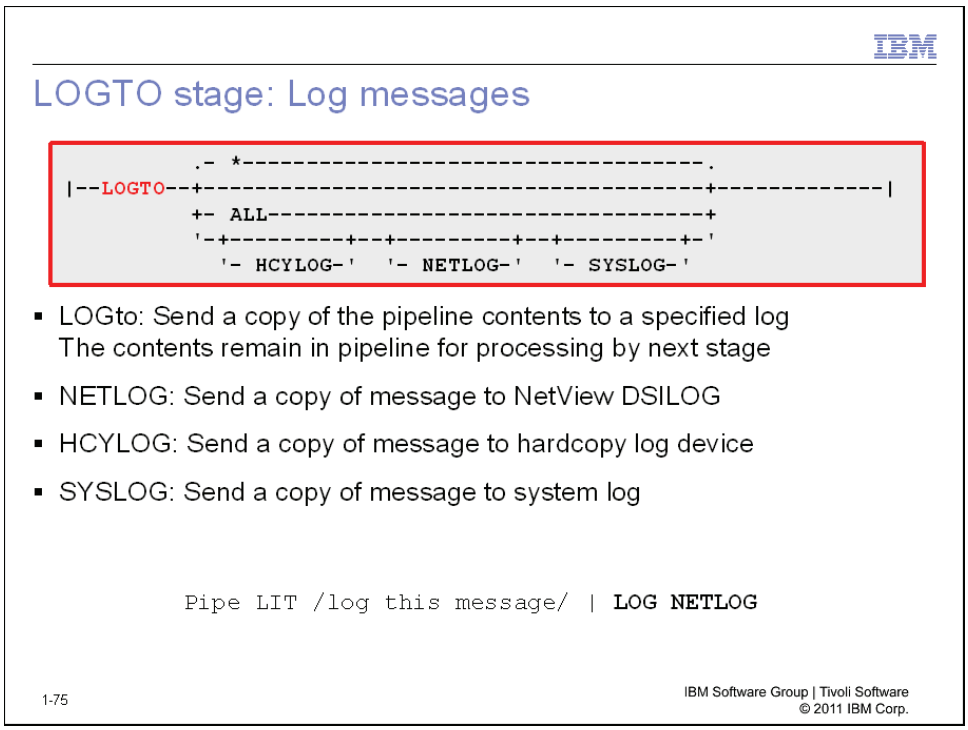
Pipeline data can be discarded, displayed, logged, or saved in NetView storage for use later by the same EXEC or a different EXEC. The CONSOLE stage can be used for displaying data to a NetView operator.

HOLE stage: Deleting all data from pipeline



The **HOLE** stage discards the contents of the pipeline. In the first example, HOLE is used for discarding the data in the pipeline after the data is saved to the stem variable. The LIT stage inserts text into the pipeline to say COMMAND COMPLETE. The second example can be used for determining if a command is *correlatable*, that it has correlated output. If the command response is displayed, it is not correlatable. You can also use the HOLE stage to discard output stream from a multiple-output pipeline.

LOGTO stage: Log messages



The **LOGTO** stage sends a copy of the contents of the pipeline to one or more of the specified logs. The input stream (unmodified) passes to the output stream for processing by the next.**LOGTO *** stage. Pipeline contents are be logged based on default settings that are defined in CNMSTYLE or with the NetView OVERRIDE and DEFAULT commands.

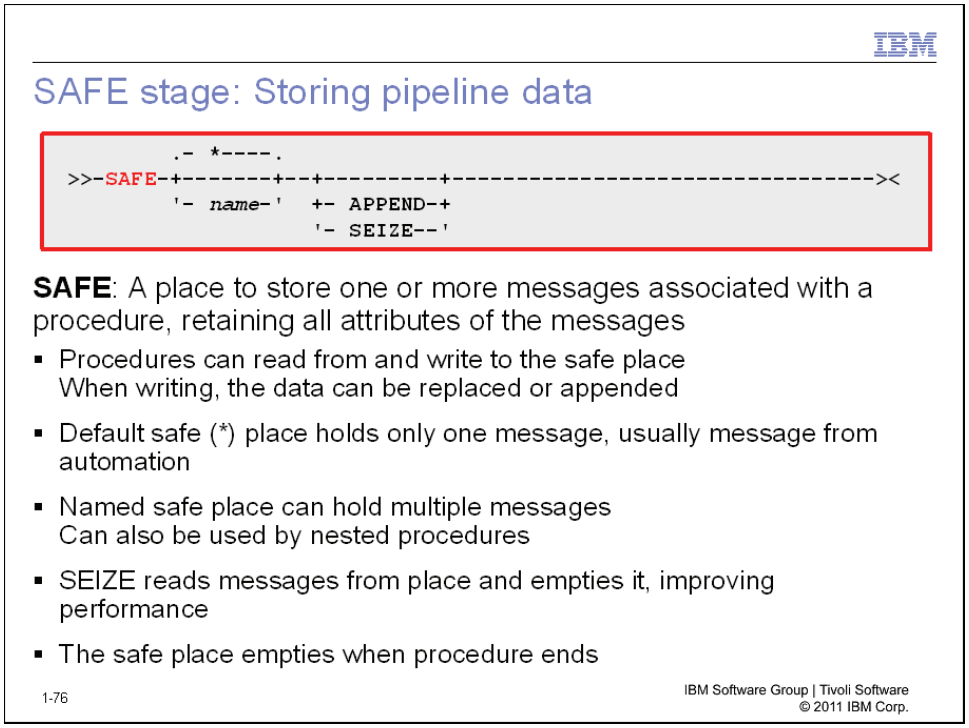
The default logging settings that you can find in CNMSTYLE are as follows:

```
DEFAULTS.NetLog = Yes
DEFAULTS.SysLog = No
DEFAULTS.HcyLog = Yes
```

You can issue a **LIST DEFAULTS** command to display the settings as follows:

```
DWO654I DISPLAY    DEFAULTS
SYSLOG: NO
NETLOG: YES
HCYLOG: YES
And so on
```

SAFE stage: Storing pipeline data



The **SAFE** stage defines a place to store one or more messages *associated with a command procedure,* for example, a REXX EXEC. With the SAFE stage, you can read from or write to a default or a named SAFE.

Note: The messages in a SAFE retain their full message structure and attributes.

When SAFE is the first stage, the specified SAFE is read into the pipeline. For a named SAFE, multiple messages could potentially go into the pipeline.

When SAFE is not the first stage, the input messages are written to the specified SAFE. For the default SAFE, just one message is written and all messages are copied to the output stream. For a named SAFE, each input message is written to the SAFE and to the output stream.

Default SAFE

The default SAFE is the current message that is associated with a command procedure. For example, when the automation table invokes a command procedure, the default safe contains the automated message. The default SAFE is preserved while the command procedure is active. The default SAFE can contain only one message.

Named SAFE

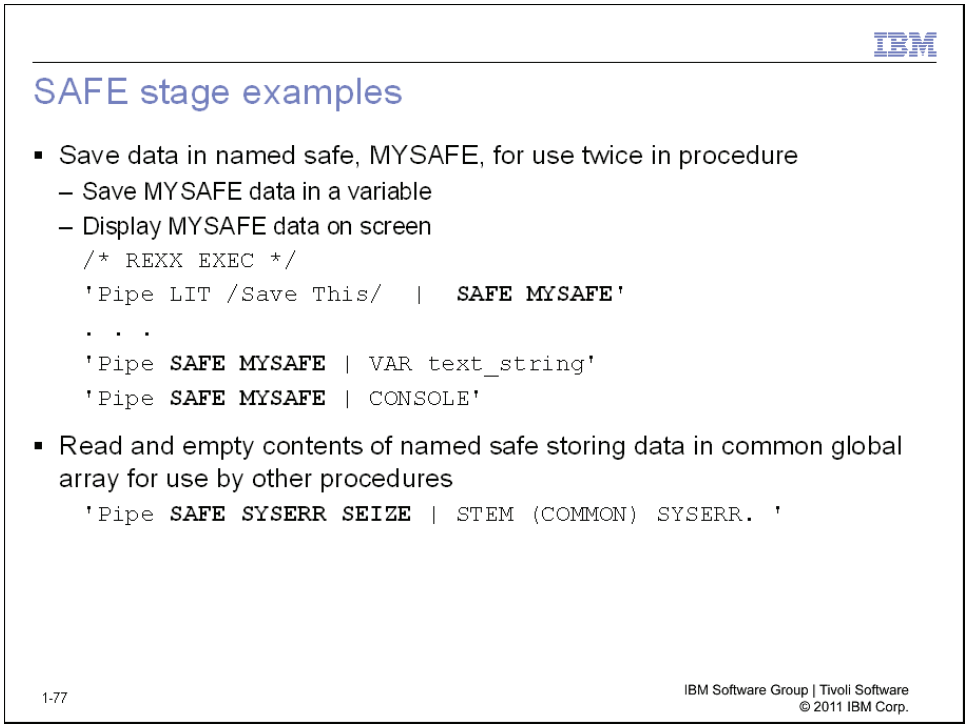
The named SAFE is a named area for a queue of messages associated with a command procedure or group of nested command procedures. For example, a REXX EXEC can write messages to a named SAFE, then call a second EXEC to read from, or write to, the named SAFE.

Note: A named SAFE is preserved while the command procedure or command group is active.

A named SAFE can contain any number of messages. A command procedure group can have any number of named SAFEs at a time.

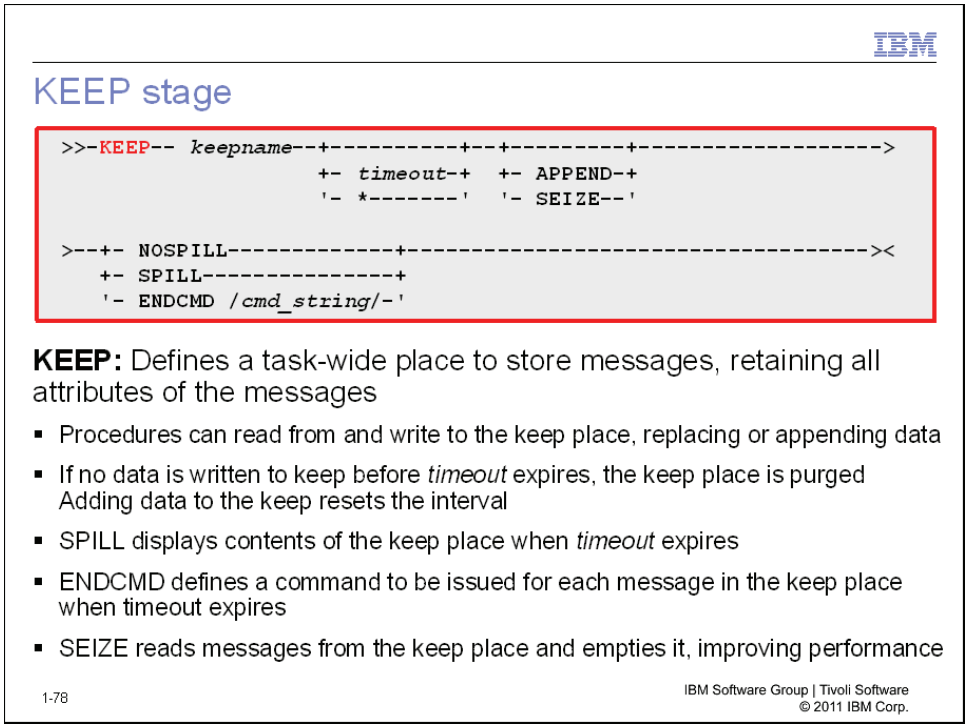
- APPEND: Specifies that data should be added after data that already exists in the named SAFE. APPEND is valid only when using a named SAFE.
- SEIZE: Use SEIZE for performance improvement when you do not need the contents of the safe to remain in the safe after a read operation.

SAFE stage examples



The second SAFE example reads the contents of the named safe, SYSERR, into the common global (stem) variable *syserr*. emptying the contents of the safe.

KEEP stage



The **KEEP** stage is similar to the SAFE stage. With the KEEP stage, you can define a task-wide place to store messages. You can read from or write to storage that the *keepname* defines. The name and message are global for the task and exist beyond the life of the procedure that creates them.

If PIPE KEEP is the first stage, it copies messages from the KEEP into the output stream. If PIPE KEEP is *not* the first stage, it copies messages from the input stream into the KEEP.

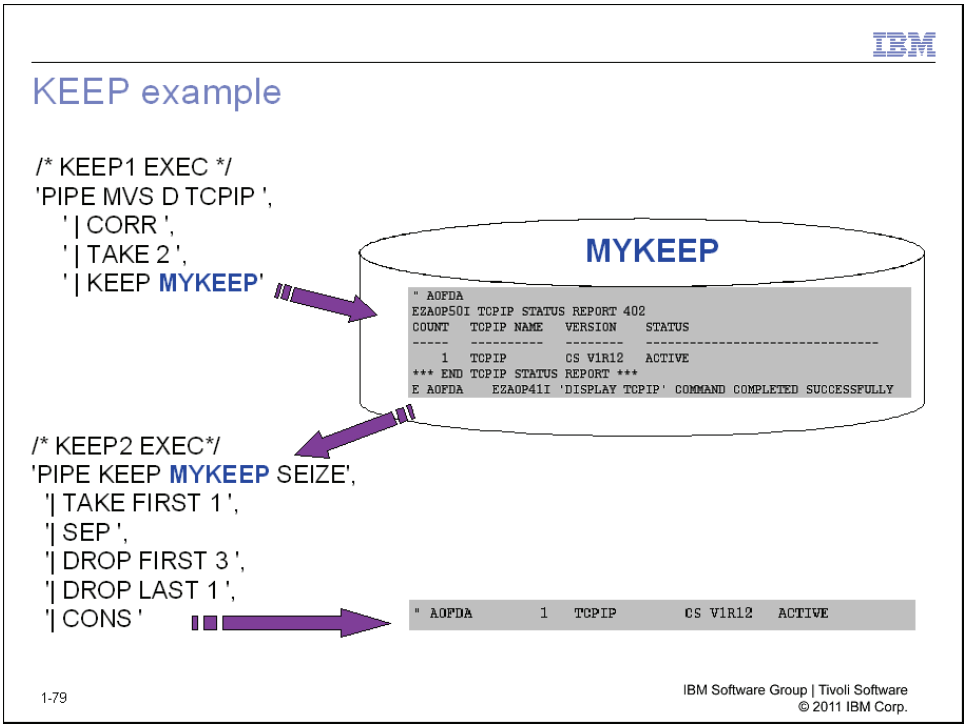
You can use the SPILL operand when KEEP is not a first stage. When the KEEP expires, SPILL indicates to display messages that are in the KEEP. The messages become subject to automation and message traps. If the KEEP expires because of a LOGOFF command or task termination, messages route to the authorized receiver.

You can use the NOSPILL operand when KEEP is not a first stage. When the KEEP expires, NOSPILL indicates to discard messages that are in the KEEP.

Use the **QRYKEEP** command to display keep statistics as follows:

```
BNH560I KEEP Status for NETOP1
Keep Name     Number Messages   Total Storage     Time Out
KEEP1           40                18212           1000
KEEP2           1                 1040            *
```

KEEP example

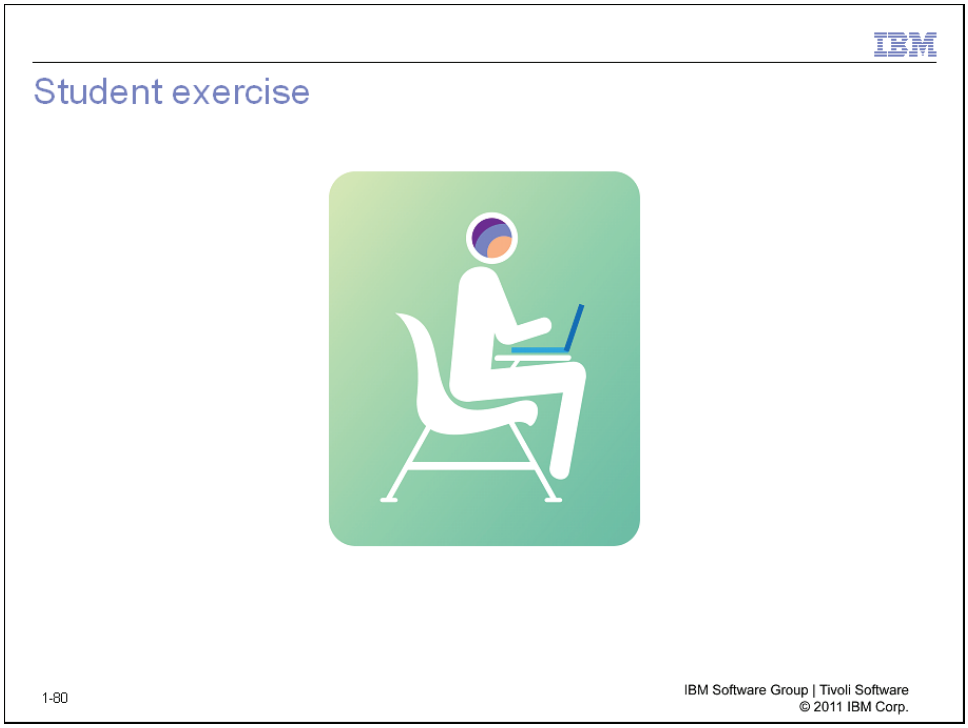


This slide shows a simple example of using a KEEP to share data between two REXX EXECS:

1. KEEP1 is a REXX EXEC that issues the MVS D TCPIP command and stores the output in a KEEP called MYKEEP. Immediately after storing the data, KEEP1 ends.

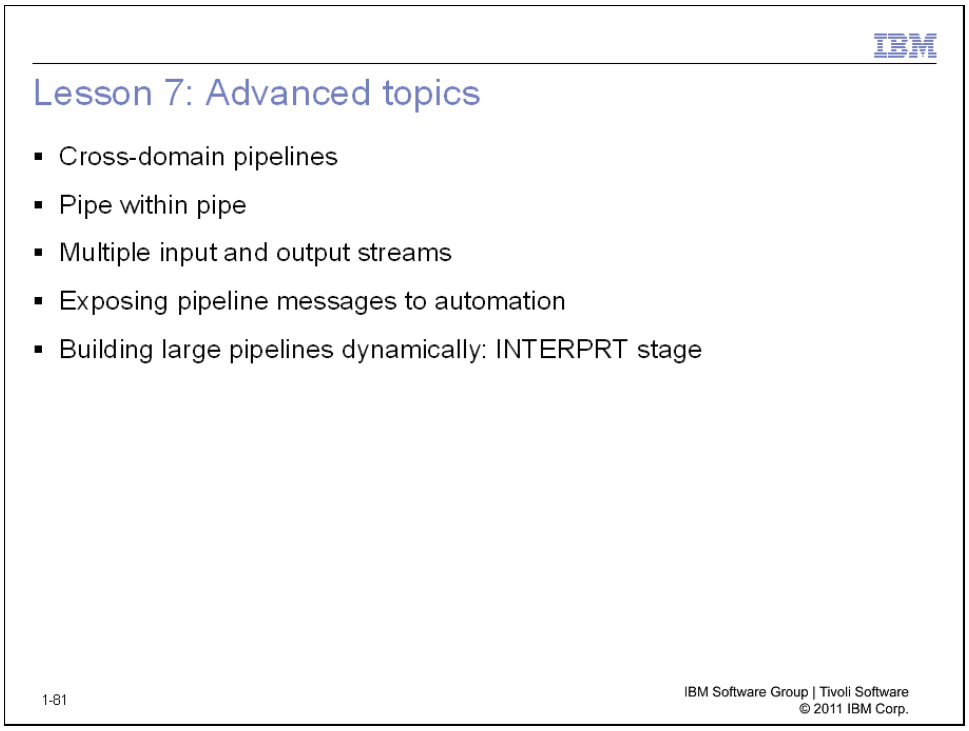
2. EXEC KEEP2 runs. KEEP2 reads the contents of MYKEEP and displays one message on the NetView console. As KEEP2 reads MYKEEP, it deletes the data from the keep with the SEIZE option.

Student exercise



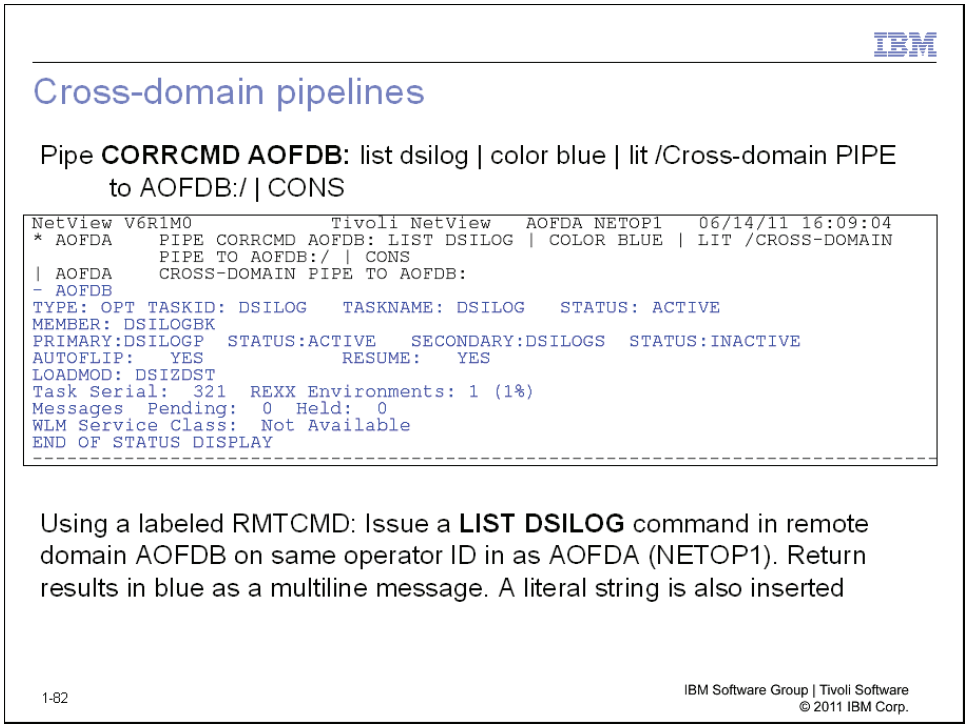
Open your *Student Exercises* book and perform Exercise 6.

Lesson 7: Advanced topics



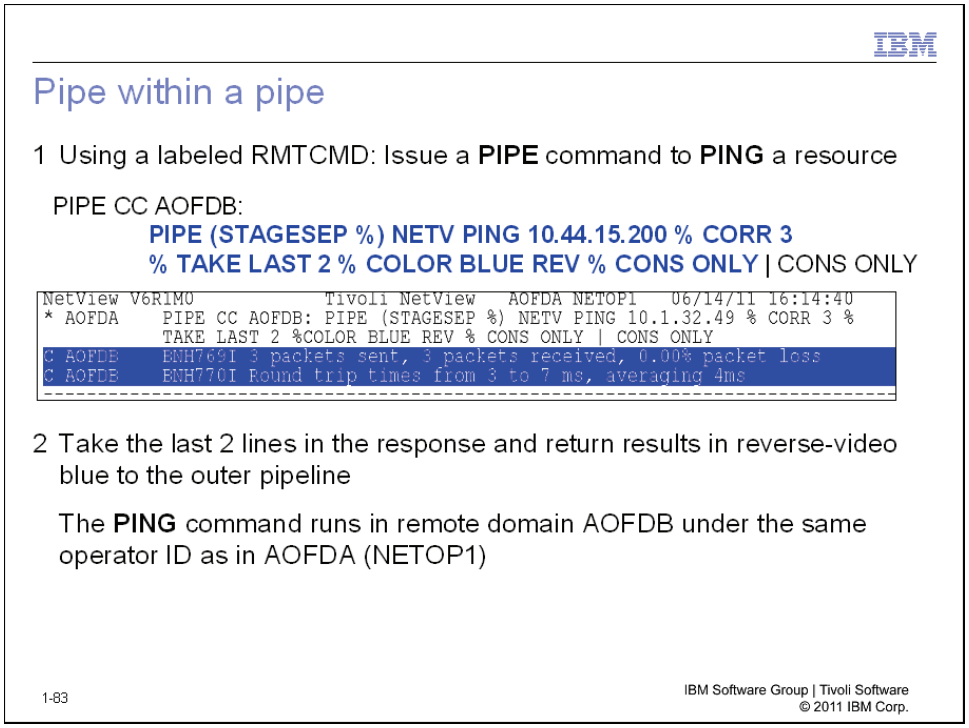
This lesson discusses advanced pipeline techniques. The INTERPRT stage is similar to a REXX INTERPRT instruction. Using the INTERPRT stage, you can dynamically build and run more complex pipelines.

Cross-domain pipelines



This example shows a labeled RMTCMD using CORRCMD to issue a LIST DSILOG in a remote NetView domain. The PIPE is issued from NETOP1 in the AOFDA domain to the AOFDB domain as the same operator. The LIST DSILOG response returns as blue text. The LITERAL string remains unchanged because it was inserted after the COLOR stage. The CORRCMD stage includes any pertinent pipeline stages from DSICCDEF.

Pipe within a pipe



This example shows how to issue a *pipe within a pipe* for sending a command to a cross-domain NetView. A *pipe within a pipe* produces performance while a second system processes data and returns only necessary responses.

A second stage separator character (%) is necessary to distinguish pipeline stages between the inner and outer pipes. Any CORRWAIT value in the inner pipe must be smaller than the CORRWAIT value in the outer pipe.

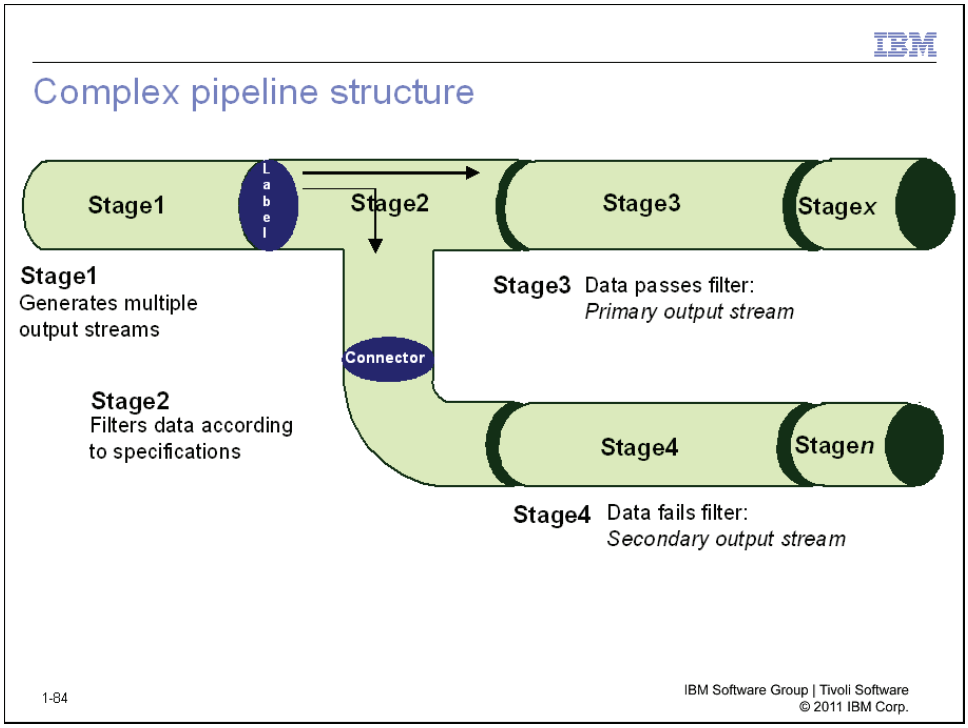
The complete ping response is as follows:

```
BNH765I Pinging tived1.tived.ibm.com at 10.1.32.49 with 3 packets
of length 16 bytes
BNH767I 16 bytes received from 10.1.32.49: seq=1 in 3ms
BNH767I 16 bytes received from 10.1.32.49: seq=2 in 7ms
BNH767I 16 bytes received from 10.1.32.49: seq=3 in 3ms
BNH769I 3 packets sent, 3 packets received, 0.00% packet loss
BNH770I Round trip times from 3 to 7 ms, averaging 4ms
```

This pipeline uses only the last two messages.

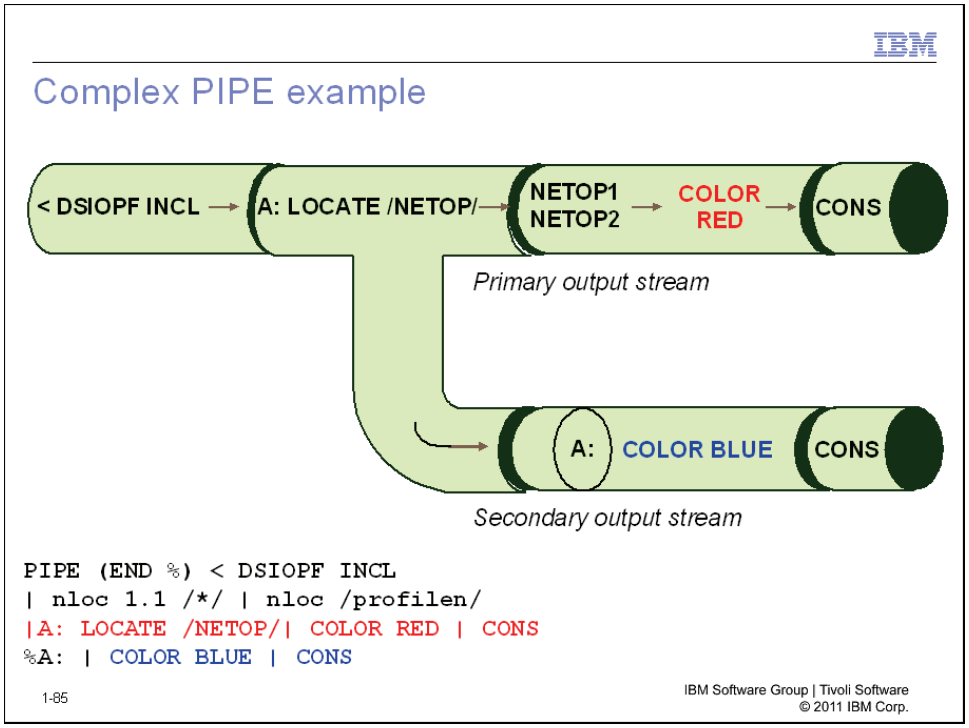
The inner pipeline ends with a CONS ONLY stage. The messages in the output stream flow to the input for the next stage of the outer pipeline, another CONS ONLY stage. CONS ONLY is necessary for both inner and outer pipelines to display the response to the PING command.

Complex pipeline structure



A complex pipeline is made up of several simple pipelines that are connected with labels. Complex pipelines are simple programs rather than complicated commands.

Complex PIPE example

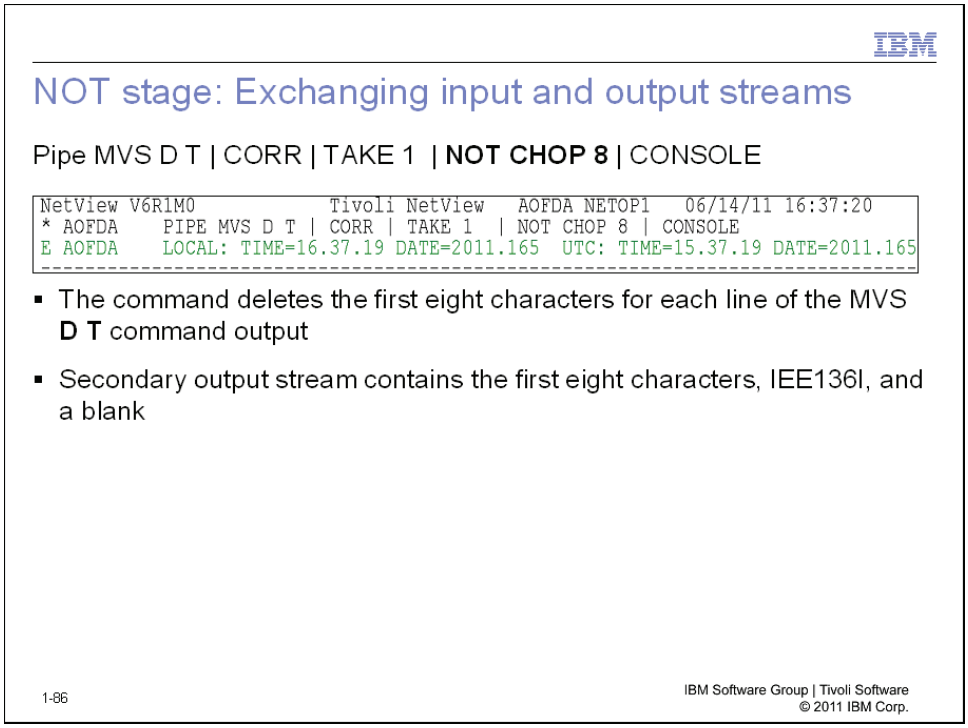


This example reads all lines that are contained in the DSIOPF file (plus included members) into the pipeline. The label **A:** on the LOCATE stage creates a *secondary* output stream for the LOCATE stage. All data in the pipeline that matches the LOCATE stage specification (inclusion of the NETOP character string) passes to the *primary* output stream. All data that does not match passes to the secondary output stream.

The COLOR stage modifies the color attribute of the records that the LOCATE stage selects to red on its primary output stream.

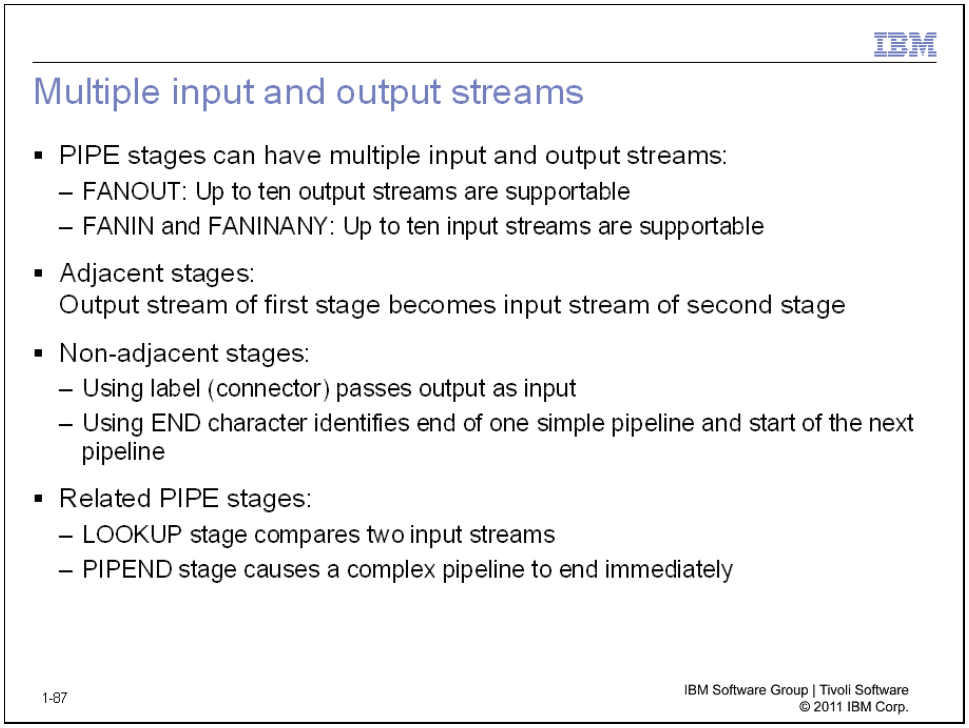
A connector labeled **%A** indicates the end of the first (simple) pipeline that is identified by the label **A:**. It indicates the target of the secondary output stage of the first pipeline LOCATE stage. All data in the secondary output stream is to change its color attribute to blue.

NOT stage: Exchanging input and output streams



You can use the **NOT** stage to exchange the input and output streams. CHOP, TAKE, TOSTRING, and LOCATE are some of the stages that can use a NOT stage.

Multiple input and output streams



When stages are adjacent each other in a pipeline, the output stream of a stage connects to the input stream of the stage that follows. When stages are *not* adjacent each other in a pipeline, labels and connectors are used for defining the input stream to the non-adjacent stages.

The *end* character indicates the end of one pipeline and the beginning of the next pipeline. The % symbol (END %) is the most common character.

A label is used for creating multiple data streams for a stage. These data streams are numbered, starting with one, and can go as high as 10, depending on the stage. When a stage is processed, the number one (primary) input stream connects to the previous stage if there is one.The number on (primary) output stream connects to the following stage if there is one.

When a connector is encountered later in the pipeline specification, a data stream is defined. The data stream connects from the stage where the label was defined to the connector. The lowest stream number available is assigned to the data stream.

If the labeled stage has an output in a simple pipeline within a complex pipeline, the data stream becomes an output from the stage that defines the label. It also becomes an input to the stage that follows the connector. If the labeled stage is an output to a stage in the pipeline specification, the data stream becomes an input to the stage that defines the label. The data steam also becomes an output to the stage that precedes the connector.

It is possible for a connector to be neither first nor last, in which case, the connector defines both an input and an output for the labeled stage. It is also possible to use two connectors in a row. This usage connects the output of one labeled stage to the input of another.

FANIN, FANINANY, and FANOUT

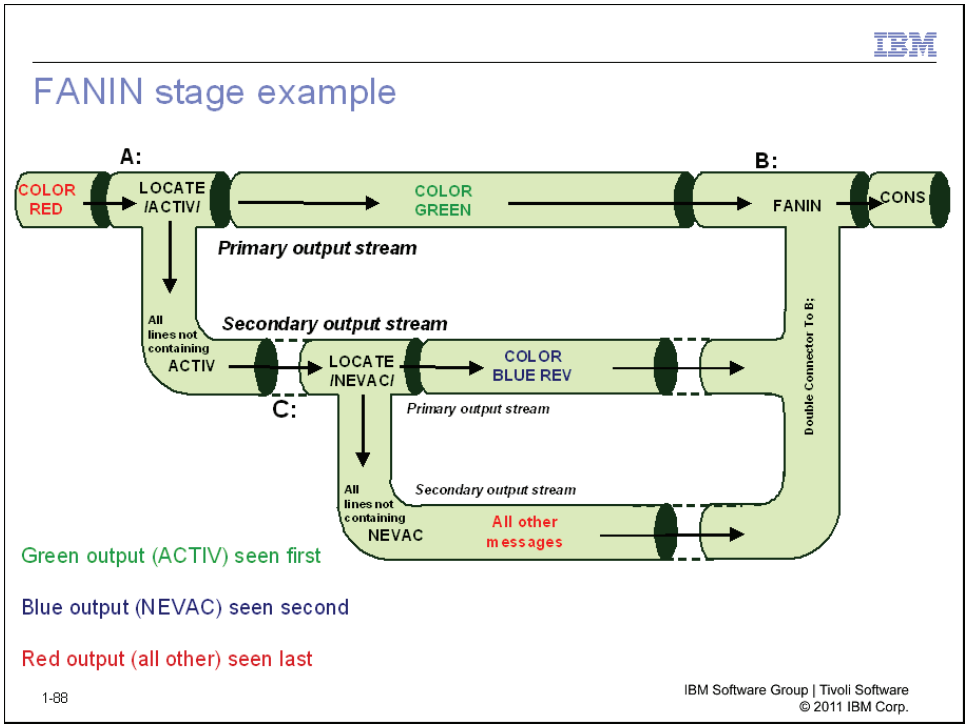
The FANIN stage reads from multiple input streams. FANIN reads from the first stream until that stream disconnects. FANIN then reads from the next input stream until it disconnects, and so on. All data that FANIN reads passes to a single output stream.

The FANINANY stage reads from each connected input stream and passes the messages to a single output stream. Messages pass in the order received regardless of their input stream.

The FANOUT stage copies the messages that it receives on its primary input stream to all connected output streams. Messages that copy to the output streams are identical except for the following cases:

- Messages that are written to the primary output stream retain the IFRAUPRI primary attribute of the original message. The copy attribute, defined by IFRAUCPY, is set to zero.
- Messages that are written to all other output streams have the IFRAUPRI primary attribute setting set to zero and the copy attribute, defined by IFRAUCPY, set to one.

FANIN stage example

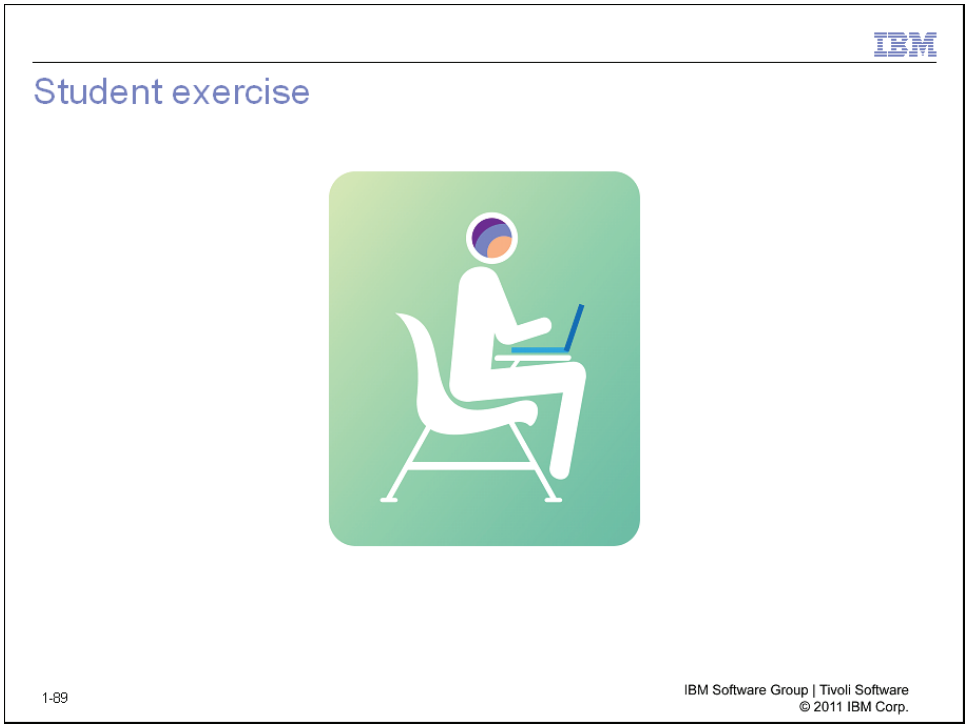


This example shows a complex pipeline where the **FANIN** stage collects all the output stages. The REXX program is as follows:

```
/*************************************************************************/
/*    Example of a complex pipeline using the FANIN stage                */
/*************************************************************************/
 MSG = 'IST482I'                  /*                                     */
'PIPE (END +) NETV CDRMS        ',/* + is the ending character of a pipe  */
   '| SEPARATE                  ',/* separate to single line messages     */
   '| LOCATE /'MSG'/            ',/* locate the IST482I message           */
   '| CHANGE /'MSG'//           ',/* discard the MSGID                    */
   '| PRESATTR RED              ',/* set color red for all lines          */
   '| A: LOCATE /ACTIV/         ',/* primary FANIN input stream (ACTIVs)  */
   '                           ',/* pass lines containing ACTIV to primary */
   '                           ',/* output, others to secondary output A:  */
   '| PRESATTR GREEN           ',/* primary output is coloured green     */
   '| B: FANIN                 ',/* read all 3 input streams             */
   '| CONSOLE                  ',/* output to console                    */
   '+ A:                       ',/* connect sec. outp. of LOCATE /ACTIV/   */
   '                           ',/* to next stage LOCATE /NEVAC/          */
   '| C: LOCATE /NEVAC/        ',/* locate all NEVACs, pass the rest to C: */
   '| PRESATTR BLUE REV        ',/* NEVACs are colored blue              */
   '| B:                       ',/* secondary FANIN input stream (NEVACs)  */
   '+ C:                       ',/* connect sec. outp. of LOCATE /NEVAC/   */
   '                           ',/* to the tertiary FANIN input Stream     */
   '| B:                      ',
    EXIT
```

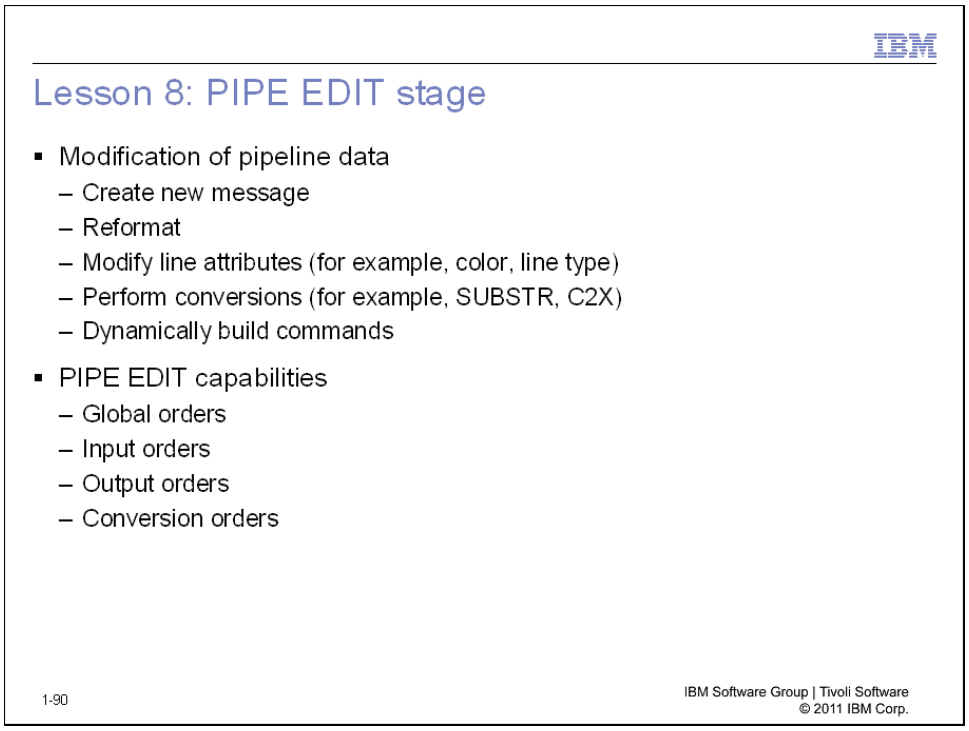
All active resources (LOCATE /ACTIV/) are to be displayed in green. All never- active resources (LOCATE /NEVAC/) are to be displayed in blue. All other resources are to be displayed in red.

Student exercise



Open your *Student Exercises* book and perform Exercise 7.

Lesson 8: PIPE EDIT stage



With the **EDIT** stage, you can make many types of changes, or edits, to a message within a pipeline. Edit data can include the following sources:

- Message data
- Line attributes
- Message attributes
- Command responses
- Literal data

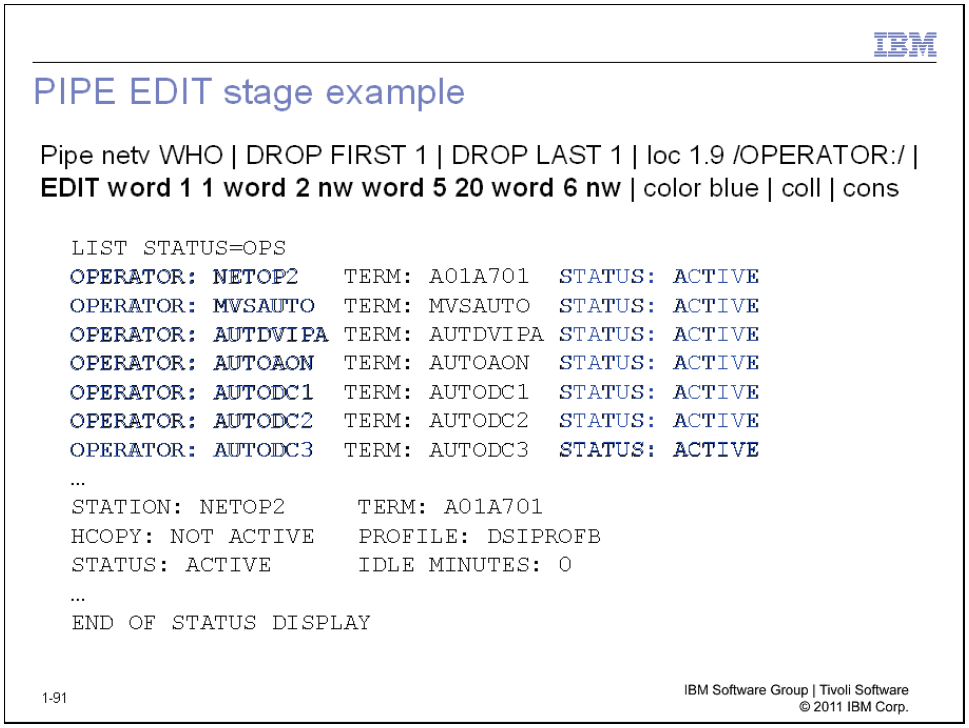
With EDIT, you can create or reformat messages. In some cases, you can modify message and line attributes. You can edit as follows:

- Prevent calling REXX programs to manipulate messages.
- Improve performance. Editing within a pipeline flow is faster than driving a command to make the changes.
- Preserve message attributes when changing the message text.
- Improve programmer productivity when writing procedures to manipulate message data.
- Dynamically create commands to issue that are based on the contents of pipeline data.

EDIT is a simple stage consisting of global orders and edit phrases. Edit phrases define the action to be taken on the data as it flows in the pipeline.

- Global orders: Define the overall environment for the subsequent edit phrases. Global orders are optional.
- Input orders: Define the source of data to be processed by the conversion and output orders of the edit phrase.
- Output orders: Define how the resulting data is to be placed in the output line and, subsequently, on the output data stream.
- Conversion orders: Define how the data is to be manipulated. Conversion orders are optional.

PIPE EDIT stage example



Explanation of PIPE EDIT stage in this example is as follows:

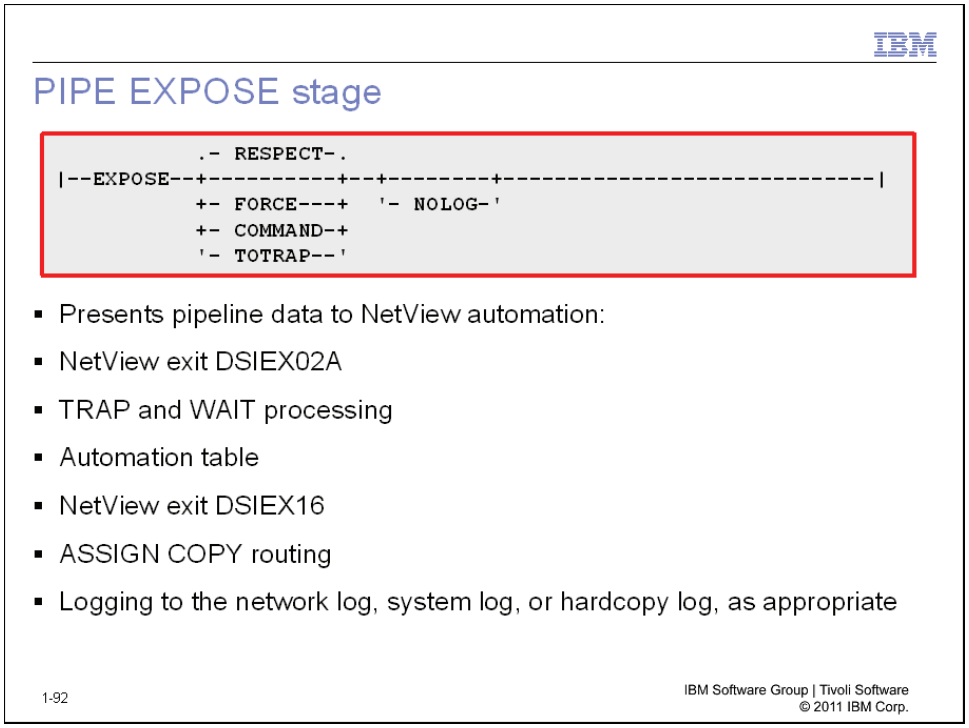
1. Issue the NetView WHO command. Delete the first and last lines in the response. Locate only the lines that begin with the string, OPERATOR:.
2. Use the PIPE EDIT stage:
   a. WORD 1 1: Take the first word (OPERATOR:) from the input stream, and write it to the first word in *position* one of the output stream.
   b. WORD 2 NW: Take the second word (for example, NETOP2) from the input stream, and write it to the *next word* of the output stream.
   c. WORD 5 20: Take the fifth word (STATUS:) from the input stream, and write it to the output stream, beginning in *position* 20.
   d. WORD 6 NW: Take the sixth word (for example, ACTIVE) from the input stream, and write it to the *next word* of the output stream.
3. Display the result in blue at the NetView operator screen. The result should be similar to the following text:

```
OPERATOR: NETOP2   STATUS: ACTIVE
OPERATOR: MVSAUTO  STATUS: ACTIVE
OPERATOR: AUTDVIPA STATUS: ACTIVE
OPERATOR: AUTOAON  STATUS: ACTIVE
OPERATOR: AUTODC1  STATUS: ACTIVE
```

```
OPERATOR: AUTODC2  STATUS: ACTIVE
OPERATOR: AUTODC3  STATUS: ACTIVE
```

And so on.

PIPE EXPOSE stage



The **EXPOSE** stage causes messages in the pipeline to be exposed to automation by passing the pipeline messages to areas as follows:

- User exit DSIEX02A
- TRAP and WAIT processing
- Automation table for possible automation
- User exit DSIEX16
- ASSIGN COPY routing
- Logging (network log, system log, or hardcopy log)

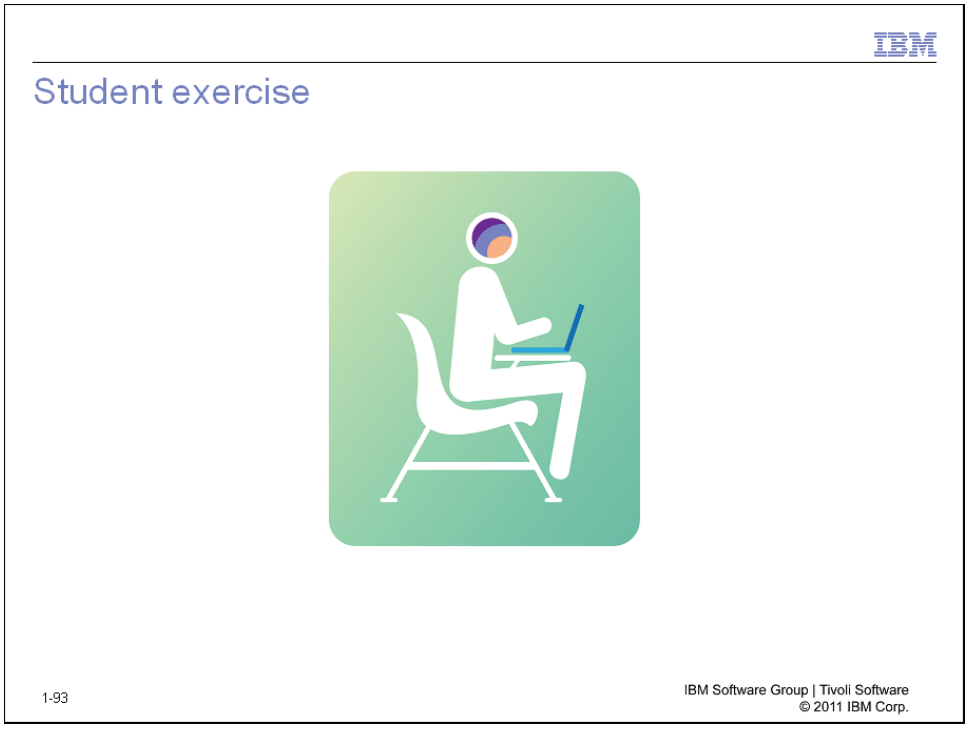
You can specify actions as follows:

- COMMAND: Specifies that messages that the processing of a command generates in a previous CORRCMD, NETVIEW, or MVS stage are to be exposed. This action is to occur before the messages are absorbed into the pipeline.
- FORCE: Specifies that messages are to be exposed to user exit 02A, message automation, and user exit DSIEX16. This action is to occur regardless of whether they have been previously exposed to those interfaces or not.
- NOLOG: Specifies that messages are to be processed as indicated by other specified keywords, but no logging is to occur.
- RESPECT: Specifies that messages that have been exposed to exit 02A, message automation, or exit 16 are not to be exposed to the same interfaces again.
- TOTRAP: Specifies that messages are to be exposed to only TRAP processing. With TOTRAP no logging occurs.

To expose the messages that the MYCLIST EXEC generates for trapping and waiting processing only, the text is as follows:

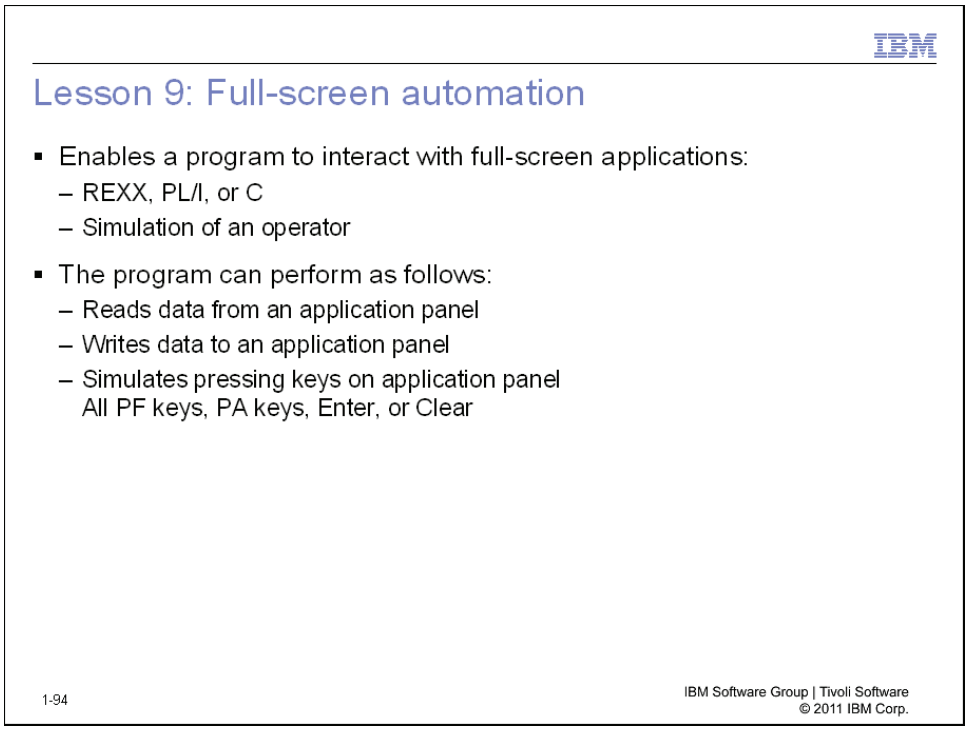
```
'PIPE NETV MYCLIST | EXPOSE TOTRAP | CONS ONLY'
```

Student exercise



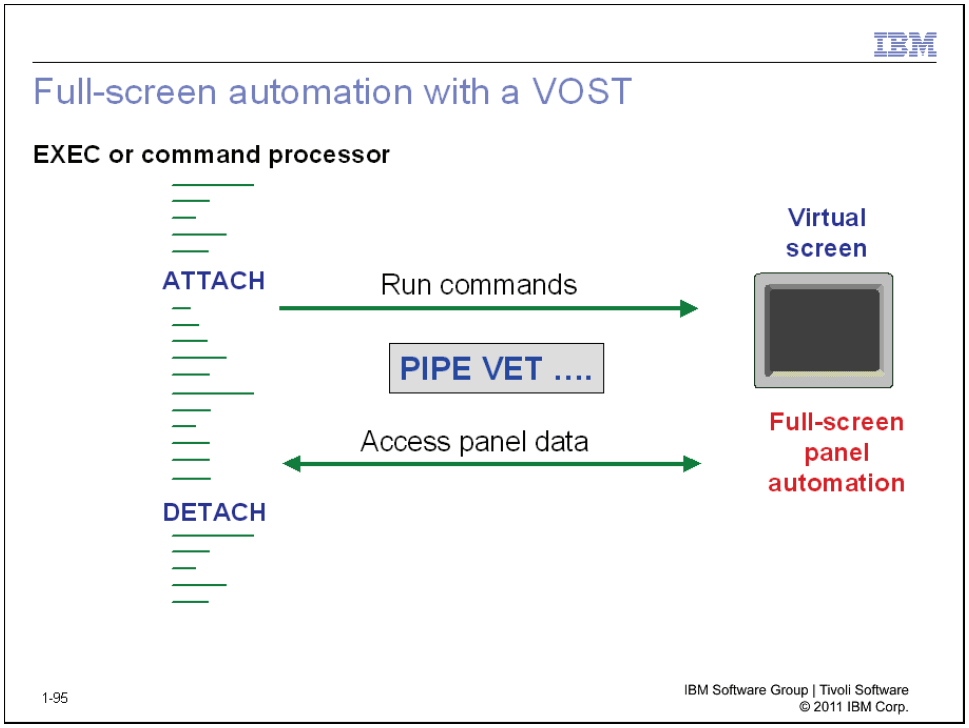
Open your *Student Exercises* book and perform Exercise 8.

Lesson 9: Full-screen automation



Using a REXX, PL/I, or C program, you can connect to a full-screen application as a simulated operator. Under the simulated operator, you can issue commands and interrogate the screen for a response. This type of automation is typically called *screen scraping*. You can use a TAF full-screen session to connect to the application.

Full-screen automation with a VOST



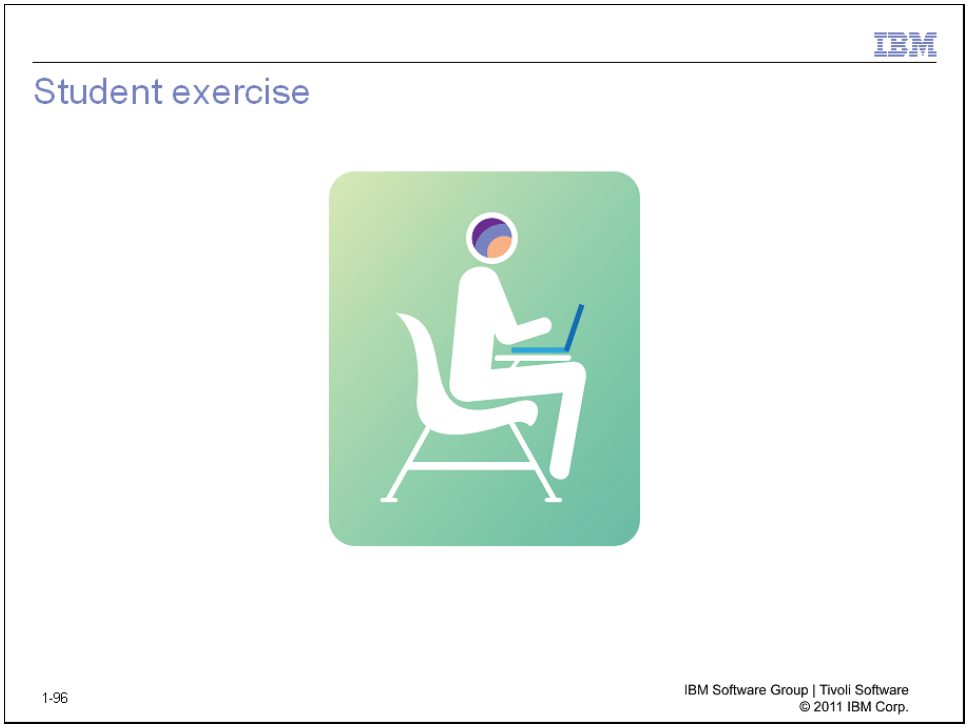
This slide provides an overview of the basic steps to implement full-screen automation.

- The **ATTACH** command is used for beginning a simulated operator session, called a virtual OST (VOST), and to issue a command in that session. The virtual screen that is created on the VOST is 24 rows by 80 characters with no query support.
- The **PIPE VET** stage is used for reading data from and writing data to a virtual screen that is associated with a virtual OST (VOST). Row and field form data returns in a multiline message, BNH150I. NetView also provides a VET command.
- The **DETACH** command is used for ending the virtual OST (VOST) that an ATTACH command creates. DETACH simulates sending a LOGOFF command to the application that runs on the VOST.

See the *IBM Tivoli NetView for z/OS 6.1 Programming: Pipes* manual for more details. Online help is also available for ATTACH, DETACH, and PIPE VET commands.

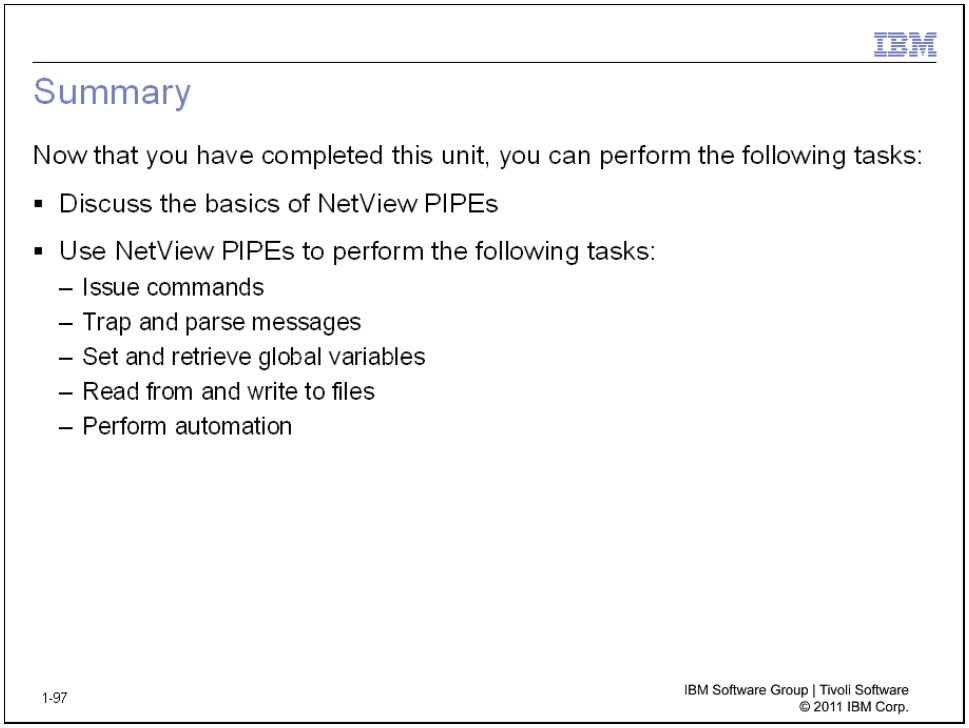
NetView also provides examples of coding full-screen automation. Browse CNMS1101 in CNMSAMP or CNME2011 in CNMCLST to see sample REXX code.

Student exercise



Open your *Student Exercises* book and perform Exercise 9.

Summary



IBM Tivoli certification and training

In today's global business world, enhancing and maintaining skills is essential to keeping pace with rapidly changing technologies. Businesses need to maximize technology potential and employees need to keep up to date with the latest information. Training and professional certification are two powerful solutions.

Certification

There are many reasons for certification:

- You demonstrate value to your customer through increased overall performance with shorter time cycles to deliver applications.
- Technical certifications assist technical professionals to obtain more visibility to potential customers.
- You differentiate your skills and knowledge from other professionals and stand out as the committed technical professional in today's competitive global world.

Online certification paths are available to guide you through the process for achieving certification in many IBM Tivoli areas. See **ibm.com/tivoli/education** for more information.

Special offer for having taken this course

Now through 31 December 2011: For having completed this course, you are entitled to a 15% discount on your next examination at any Thomson Prometric testing center worldwide. Use this special promotion code when registering online or by telephone to receive the discount: **15CSWR**. (This offer might be withdrawn. Check with the testing center as described later in this section.)

Role-based certification

All IBM certifications are based on job roles. They focus on a job a person must do with a product, not just the product's features and functions. Tivoli Professional Certification uses the following job roles used:

- IBM Certified Advanced Deployment Professional
- IBM Certified Deployment Professional
- IBM Certified Administrator
- IBM Certified Solution Advisor
- IBM Certified Specialist
- IBM Certified Operator

Training

A broad spectrum of courses, delivery options, and tools helps keep your employees up to date with the latest IBM Tivoli information:

- *Instructor-led training (ILT)*
  Live interaction with an IBM instructor, hands-on lab exercises, and networking with your peers from other companies
  ibm.com/tivoli/education
- *Instructor-led online (ILO)*
  All the benefits of ILT, but savings on travel dollars and training costs
  ibm.com/training/ilo
- *Self-paced virtual classes (SPVC)*
  Interactive and hands-on exercises on your schedule
  ibm.com/training/us/spvc
- *Web-based training (WBT)*
  Training anywhere, any time, that saves you money and travel
  ibm.com/training/us/tivoli/wbt
- *Multimedia library*
  Modules supporting new and experienced learners with fully animated multimedia clips, step-by-step audio, and companion text
  ibm.com/software/tivoli/education/multimedialibrary
- *IBM Education Assistant*
  More specific, granular web-based training with individual presentations on specific topics
  www-01.ibm.com/software/info/education/assistant/
- *Corporate Education Licensing Program (CELP)*
  Solutions for large IBM customers who need to adopt IBM Tivoli's tools and technologies
  ibm.com/training/us/tivoli/celp
- *Tivoli training paths*
  Course maps with flow charts and course descriptions to help you find the right course
  ibm.com/training/us/tivoli/paths

Printed in Ireland

CT6E5EN