

TIBCO WebFOCUS®

App Studio Maintain Data Language Reference

Release 8.2 Version 04 and Higher

March 2021

DN4501618.0321



Contents

1. Language Rules Reference	11
Case Sensitivity	12
Specifying Names	13
Reserved Words	15
What Can You Include in a Procedure?	16
Multi-Line Commands	17
Terminating Command Syntax	17
Adding Comments	18
2. Expressions Reference	21
Types of Expressions	21
Expressions and Variable Formats.	23
Writing Numeric Expressions	23
Order of Evaluation.	25
Evaluating Numeric Expressions.	25
Identical Operand Formats.	26
Different Operand Formats.	27
Continental Decimal Notation.	27
Writing Date Expressions	28
Formats for Date Values.	29
Evaluating Date Expressions.	29
Selecting the Format of the Result Variable.	30
Manipulating Dates in Date Format.	31
Using a Date Constant in an Expression.	31
Extracting a Date Component.	31
Combining Variables With Different Components in an Expression.	31
Different Operand Date Formats.	32
Using Addition and Subtraction in a Date Expression.	33
Writing Date-Time Expressions	34
Manipulating Date-Time Values Directly.	36
Comparing and Assigning Date-Time Values.	36
Date-Time Subroutines.	37
Writing Character Expressions	39

Concatenating Character Strings.....	40
Evaluating Character Expressions.....	40
Variable-Length Character Variables.....	42
Writing Logical Expressions	44
Relational Expressions.....	44
Boolean Expressions.....	44
Evaluating Logical Expressions.....	45
Writing Conditional Expressions	46
Handling Null Values in Expressions	47
Assigning Null Values: The MISSING Constant.....	47
Conversion in Mixed-Format Null Expressions.....	48
Testing Null Values.....	48
3. Command Reference	51
Language Summary	51
Defining a Procedure.....	51
Defining a Maintain Data Function (a Case).....	51
Defining Blocks of Code.....	52
Transferring Control.....	52
Executing Procedures.....	52
Using Loops.....	53
Using Forms.....	53
Defining Classes.....	53
Creating Variables.....	53
Assigning Values.....	53
Manipulating Stacks.....	53
Selecting and Reading Records.....	54
Conditional Actions.....	55
Writing Transactions.....	56
Setting Reporting Server Parameters.....	57
Using Libraries of Classes and Functions.....	57
Messages and Logs.....	57
BEGIN	58

CALL59

CASE 63

 Calling a Function: Flow of Control.....65

 Passing Parameters to a Function.....65

 Using the Return Value of a Function..... 66

 Using the Top Function..... 66

COMMIT67

COMPUTE 68

 Using COMPUTE to Call Functions.....72

 Using COMPUTE to Dynamically Change the Property of an Object..... 72

COPY 73

DECLARE77

 Local and Global Declarations..... 80

DELETE 80

DESCRIBE84

 Class Member Functions.....87

 Defining and Using Superclasses and Subclasses.....93

END 96

EXEC 96

FocCount98

FocCurrent98

FocError98

FocErrorRow 99

FocFetch 99

FocIndex 99

FocMsg 100

GOTO101

 Using GOTO With Data Source Commands..... 103

 GOTO and ENDCASE..... 104

 GOTO and PERFORM..... 104

IF 104

 Coding Conditional COMPUTE Commands..... 107

INCLUDE107

Data Source Position.	110
Null Values.	111
INFER	111
Defining Non-Data Source Columns.	113
MAINTAIN	113
Specifying Data Sources With the MAINTAIN Command.	115
Calling a Procedure From Another Procedure.	115
MATCH	116
How the MATCH Command Works.	118
MODULE	118
What You Can and Cannot Include in a Library.	119
NEXT	119
Copying Data Between Data Sources.	122
Loading Multi-Path Transaction Data.	122
Retrieving Multiple Rows: The FOR Phrase.	123
Using Selection Logic to Retrieve Rows.	123
Using NEXT After a MATCH.	126
Using NEXT for Data Source Navigation: Overview.	126
Data Source Navigation: Using NEXT With One Segment.	128
Data Source Navigation: Using NEXT With Multiple Segments.	129
Data Source Navigation: Using NEXT Following NEXT or MATCH.	131
Unique Segments.	133
ON MATCH	134
ON NEXT	135
ON NOMATCH	135
ON NONEXT	136
PERFORM	137
Using PERFORM to Call Maintain Data Functions.	138
Using PERFORM With Data Source Commands.	139
Nesting PERFORM Commands.	139
Avoiding GOTO With PERFORM.	139
REPEAT	139
Branching Within a Loop.	145

REPOSITION	146
REVISE	147
ROLLBACK	150
DBMS Combinations	151
SAY	151
Writing Segment and Stack Values	152
Choosing Between the SAY and TYPE Commands	152
SET	152
showLayer	154
STACK CLEAR	154
STACK SORT	155
Sorting Data With the Using CASE_INSENSITIVE Parameter	156
SYS_MGR	157
SYS_MGR.DBMS_ERRORCODE	157
SYS_MGR.ENGINE	158
SYS_MGR.FOCSET	159
SYS_MGR.GET_INPUTPARAMS_COUNT	161
SYS_MGR.GET_NAMEPARM	162
SYS_MGR.GET_POSITIONPARM	163
SYS_MGR.PRE_MATCH	163
TYPE	165
Including Variables in a Message	166
Embedding Horizontal Spacing Information	166
Embedding Vertical Spacing Information	166
Coding Multi-Line Message Strings	167
Justifying Variables and Truncating Spaces	167
Writing Information to a File	168
UPDATE	168
Update and Transaction Variables	171
Data Source Position	171
Unique Segments	172
Winform	172
Displaying Default Values in a Form	173

WINFORM SET	174
A. Ensuring Transaction Integrity	175
Transaction Integrity Overview	175
Why Is Transaction Integrity Important?	176
Defining a Transaction	177
When Does a Data Source Command Cause a Transaction to Fail?	178
Canceling a Transaction	178
Transactions and Data Source Position	179
How Large Should a Transaction Be?	179
Designing Transactions That Span Procedures	179
Designing Transactions That Span Data Source Types	181
Designing Transactions in Multi-Server Applications	181
When an Application Ends With an Open Transaction	181
Evaluating Whether a Transaction Was Successful	182
Concurrent Transaction Processing	182
Ensuring Transaction Integrity for FOCUS Data Sources	184
Setting COMMIT	184
Sharing Access to FOCUS Data Sources	185
How the FOCUS Database Server and Change Verification Work	186
Selecting Which Segments Will Be Verified for Changes	187
Identifying the FOCUS Database Server	188
Using Report Procedures and a FOCUS Database Server	189
Accessing Report Procedures When Using a FOCUS Database Server	190
Sharing Data Sources With Legacy MODIFY Applications	191
Ensuring Transaction Integrity for DB2 Data Sources	191
Using Transaction Locking to Manage DB2 Row Locks	193
Using Change Verification to Manage DB2 Row Locks	195
B. Developing Classes and Objects	199
What Are Classes and Objects?	199
Class Properties: Member Variables and Member Functions	200
Inheritance: Superclasses and Subclasses	202
Defining Classes	202

Reusing Classes: Class Libraries 209

Declaring Objects 210

C. MNTCON Commands 213

 MNTCON CDN_FEXINPUT 213

 MNTCON COMPILE 215

 MNTCON EX 215

 Invoking Maintain Procedures: Passing Parameters..... 217

 MNTCON EXIT_WARNING 218

 MNTCON MATCH_CASE 218

 MNTCON RADIO_BUTTON_EMIT_TEXT 220

 MNTCON REMOTESTYLE 220

 MNTCON RUN 221

Legal and Third-Party Notices 225

Language Rules Reference

You can use the App Studio Maintain Data language more effectively if you are familiar with its standards, including:

- When to use uppercase and lowercase characters.
- When to spell out keywords in full.
- How to name fields, functions, and other procedure components.
- Which words to avoid using as names of procedure components.
- What sort of components you can include in a procedure.
- How to continue a command onto additional lines.
- How to terminate command syntax.
- How to include comments in a procedure.

Data source descriptions and App Studio procedures are not part of the Maintain Data language and are subject to different rules. The language rules for data source descriptions are discussed in the *Describing Data With WebFOCUS Language* manual. The language rules for App Studio procedures are discussed in the *Developing Reporting Applications* manual.

In this chapter:

- [Case Sensitivity](#)
 - [Specifying Names](#)
 - [Reserved Words](#)
 - [What Can You Include in a Procedure?](#)
 - [Multi-Line Commands](#)
 - [Terminating Command Syntax](#)
 - [Adding Comments](#)
-

Case Sensitivity

By default, Maintain Data does not usually distinguish between uppercase and lowercase letters. You can enter keywords and names, such as data source and field names, in any combination of uppercase and lowercase letters. The only two exceptions are the MAINTAIN and END keywords used to begin and end a request. These keywords must be uppercase.

However, if mixed-case or NLS field and segment names are used in your application by enabling the MNTCON MATCH_CASE command, you must be consistent with the case style used in the names you give your variables and other application components. In addition, function names must match exactly as documented. However, Maintain Data keywords, such as Repeat and Include, do not need any special consideration when using this feature. For more information on MNTCON MATCH_CASE, see [MNTCON MATCH_CASE](#) on page 218.

For example, the following ways of specifying the REPEAT command are equally valid, and Maintain Data always considers them to be identical:

REPEAT

repeat

RePeat

REPeat

By default, you can mix uppercase and lowercase to make variable names more understandable to a reader. For example, the stack name SALARYSTACK could also be represented as SalaryStack.

Note: In this content, sample Maintain Data source code shows keywords in uppercase, and user-defined names, such as field and stack names, in mixed-case. This is only a document convention, not a Maintain Data language rule. As already explained, you can code Maintain Data commands in uppercase and lowercase.

While, by default, Maintain Data is not sensitive to the case of syntax, it is sensitive to the case of data. For example, the MATCH command distinguishes between the values SMITH and Smith.

Note: Any Master Files that Maintain Data accesses must have field and segment names in uppercase, unless the MATCH_CASE feature is enabled.

Specifying Names

Maintain Data offers you flexibility when naming and referring to procedure components, such as fields, functions, Winform buttons, and stacks. When naming a component, be aware of the following guidelines:

- ❑ **Length of names.** Unqualified names that are defined in a Maintain Data procedure (such as the unqualified names of Winforms, functions, and stacks) can be up to 66 characters long.

There is no limit on the length of a qualified name, as long as the length of each of its component unqualified names does not exceed 66 characters.

Master File names, and names defined within a Master File (such as names of fields and segments), are subject to standard Master File language conventions, as defined in the *Describing Data With WebFOCUS Language* manual.

Procedure name length is dependent on the operating system.

- ❑ **Valid characters in a name.** All names must begin with a letter, and can include any combination of letters, numbers, and underscores (_).
- ❑ **Identical names.** Most types of items in a Maintain Data procedure can have the same name, but this is not recommended. Data sources, stacks, and Winforms cannot have the same name within the same Maintain Data procedure.

For example, you may give the same name to fields in different segments, data sources, and stacks, and to controls in different Winforms, as long as you prevent ambiguous references by qualifying the names. A data source, a stack, and a Winform used in the same procedure can never have the same name.

- ❑ **Qualified names.** In general, whenever you can qualify a name, you should do so.

Maintain Data requires that the qualification character be a period (.). The QUALCHAR parameter of the SET command must therefore be set to the default.

If a qualified name cannot fit onto the current line, you can break the name at the end of any one of its components, and continue it onto the next line. The continued name must begin with the qualification character. In the following example, the continued line is indented for clarity:

```
FOR ALL NEXT ThisIsAVeryLongDataSourceName.ThisIsAVeryLongSegmentName
    .ThisIsAVeryLongFieldName INTO CreditStack;
```

You can qualify the names of:

- ❑ **Controls.** You can qualify a control name with the name of the Winform in which it is found. For example, if a button named UpdateButton is in a form named CreditForm, you could refer to the button as:

`CreditForm.UpdateButton`

- ❑ **Member functions and member variables.** When referring to the member functions and member variables of an object, you should always use the fully-qualified name of the function or variable (that is, the name in the Winform *objectname.functionname* or *objectname.variablename*).
- ❑ **Fields and columns.** You can qualify a variable name with the name of the data source, segment, and/or stack in which it is found, using a period (.) as the qualification character.

Qualification is important when:

- ❑ You are working with two or more data sources in one Maintain Data procedure, and the data sources have field names in common.
- ❑ A field is present in both a data source and a stack, but it is not clear from the context which one is being referenced.

For example, both the Employee and JobFile data sources have a field named JobCode. If you want to issue a NEXT command for the JobCode field in Employee, you would use a qualified field name:

```
NEXT Employee.JobCode;
```

You can qualify a field name with any combination of its data source, segment, and stack names. When including a stack name, you have the option of specifying a particular row in the stack. If you use several qualifiers, they must conform to the following order:

stackname(row).datasourcename.segmentname.fieldname

If you refer to a field using a single qualifier, such as Sales in the following example, and the qualifier is the name of both a segment and a stack, Maintain Data assumes that the name refers to the stack. To refer to the segment in this case, use the data source qualifier.

```
Sales.Quantity
```

- ❑ **Truncated names.** You must spell all field names in full. Maintain Data does not recognize truncated names, such as Dep for a field named Department.

- ❑ **Name aliases.** You cannot refer to a field by its alias in a Maintain Data procedure. (An alias is defined by the ALIAS attribute of a field in a Master File.)

Reserved Words

The words in the following table are Maintain Data keywords and are reserved. You may not use them as identifiers. Identifiers are names of application components (such as, but not limited to, classes, functions, data sources, data source segments, stacks, stack columns, scalar variables, and Winforms).

In addition to these words, you may not use the names of built-in functions to name functions that you create yourself. See the *Using Functions* manual for a complete list of built-in functions.

If a procedure uses an existing Master File that employs a reserved word as a field name, you can refer to the field by qualifying its name with the name of the segment or data source.

Note: All reserved words can be written in any case, even when using MNTCON MATCH_CASE ON.

ALL	AND	AS	ASK	AT
BEGIN	BIND	BY	CALL	CASE
CFUN	CLASS	CLEAR	COMMIT	COMPUTE
CONTAINS	CONTENTS	COPY	current	DATA
DECLARE	DECODE	DELETE	DEPENDENTS	DESCRIBE
DFC	DIV	DROP	DUMP	ELSE
END	ENDBEGIN	ENDCASE	ENDDESCRIBE	ENDREPEAT
EQ	EQ_MASK	ERRORS	EVENT	EXCEEDS
EXEC	EXECSQL	EXIT	EXITREPEAT	EXPORT
FALSE	FILE	FILES	FIND	FocCount
FocCurrent	FocEnd	FocEndCase	FocEOF	FocError
FocErrorRow	FocIndex	FOR	FROM	GE

What Can You Include in a Procedure?

GET	GOTO	GT	HERE	HIGHEST
HOLD	IF	IN	IMPORT	INCLUDE
INFER	INTO	IS	IS_LESS_THAN	IS_NOT
KEEP	LE	LIKE	LT	MAINTAIN
MATCH	MISSING	MOD	MODULE	MOVE
NE	NE_MASK	NEEDS	NEXT	NO
NOT	NOWAIT	OBJECT	OF	OFF
OMITS	ON	OR	PATH	PERFORM
QUIT	REPEAT	REPOSITION	RESET	RETURN
RETURNS	REVISE	ROLLBACK	SAY	SELECTS
self	SET	SOME	SORT	SQL
STACK	TAKES	THEN	TO	TOP
TRIGGER	TRUE	TYPE	UNLIKE	UNTIL
UPDATE	WAIT	WHERE	WHILE	WINFORM
XOR	YES	YRT		

What Can You Include in a Procedure?

You can include the following items in a Maintain Data procedure:

- Maintain Data language commands.** All Maintain Data commands must be located within a Maintain Data function, except for the MAINTAIN, MODULE, DESCRIBE, CASE, and END commands, as well as global DECLARE commands, all of which must be located outside of a function. In the Maintain Data Editor, commands are displayed in blue by default. For more information, see [Command Reference](#) on page 51.
- Comments.** In the Maintain Data Editor, comments are displayed in green by default. For more information, see [Adding Comments](#) on page 18.
- Blank lines.** Add blank lines to separate functions and other logic so that the procedure is easier to read.

If a Maintain Data procedure is a starting procedure (sometimes known as a root procedure), and it is not called by any other Maintain Data procedures, it can also contain Dialogue Manager commands preceding the MAINTAIN command. Dialogue Manager commands are described in the *Developing Reporting Applications* manual.

Multi-Line Commands

You can continue almost all Maintain Data commands onto additional lines. The continued command can begin in any column, and can be continued for any number of lines.

The only exceptions are the TYPE command, which uses a special convention for continuing, and the beginning of the REPEAT command, which cannot be continued.

In the following example, all continued lines are indented for clarity:

```
MAINTAIN FILES VideoTrk
    AND Movies
.
.
.
IF CustInfo.FocIndex GT 1
    THEN COMPUTE CustInfo.FocIndex = CustInfo.FocIndex - 1;
    ELSE COMPUTE CustInfo.FocIndex = CustInfo.FocCount;
```

Terminating Command Syntax

When you code a Maintain Data command, you terminate its syntax using one of the following:

- ❑ **A semicolon (;).** For most commands that can be terminated with a semicolon, the semicolon is optional. Even when it is optional, supplying it is recommended.

Coding suggestion: Supplying optional semicolons is preferable, because if you omit them, when you invoke functions in that procedure, you must do so using the COMPUTE or PERFORM commands. By supplying optional semicolons in a procedure, you can invoke functions more directly, by simply specifying their names. Supplying optional semicolons is also preferable because if you supply them in a procedure, you can code assignment statements more succinctly by omitting the COMPUTE keyword.

For example, the following NEXT command, assignment statement, and invocation of the DisplayEditForm function are all terminated with semicolons:

```
FOR ALL NEXT CustID INTO CustOrderStack;
EditFlag = CustOrderStack().Status;
DisplayEditForm();
```

- ❑ **An end keyword.** Some commands, such as BEGIN, CASE, and REPEAT, bracket a block of code. You indicate the end of the block by supplying the appropriate END keyword (for example, ENDBEGIN, ENDCASE, or ENDREPEAT).

In the following example, the CASE command is terminated with an ENDCASE keyword:

```
CASE UpdateAcct
UPDATE SavingsAcct FROM TransactionStack;
IF FocError NE 0 THEN TransErrorLog();
ENDCASE
```

Most commands use one of these methods (a semicolon or an end keyword) exclusively, as described for each command in [Command Reference](#) on page 51.

Adding Comments

By adding comments to a procedure, you can document its logic, making it easier to maintain. You can place a comment on its own line, at the end of a command, or even in the middle of a command. You can also place a comment in the middle of the procedure, at the very beginning of the procedure before the MAINTAIN command, or at the very end of the procedure following the END command. You can place any text within a comment.

There are two types of comments:

- ❑ **Stream comments.** Begin with `$*` and end with `*$`. Maintain Data interprets everything between these two delimiters as part of the comment. A comment can begin on one line and end on another line, and can include up to 51 lines.

For example:

```
MAINTAIN
  $* This is a stream comment *$
  TYPE "Hello world";
  $* This is a second stream comment.
  This is still inside the second comment!
  This is the end of the second comment *$
  $* Document the TYPE statement--> *$ TYPE "Hello again!"; *$ Goodbye *$
END
```

- ❑ **Line comments.** Begin with `$$` or `-*` and continue to the end of the line. For example:

```
MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO Pay;
- * This entire line is a comment.
COMPUTE Pay.NewSal/D12.2;
.
.
.
END
```

You can also place a comment at the end of a line of code:

```

MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO Pay; $$ Put root seg into a stack
COMPUTE Pay.NewSal/D12.2;
.
.
.
END

```

You can place a comment at the end of a line containing a command that continues onto the next line:

```

MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO Pay  -* Put root seg into a stack
WHERE Department IS 'MIS';
COMPUTE Pay.NewSal/D12.2;
.
.
.
END

```

You can include all types of comments in the same procedure:

```

MAINTAIN
  TYPE "Hello world"; -* This is a TYPE command
  $* This is a stream comment
    that runs onto a second line *$
*$ Document the TYPE statement--> *$ TYPE "Hello again!"; $$ Goodbye
.
.
.
END

```

While Maintain Data uses the same comment characters (-*) as Dialogue Manager, it is only in a Maintain Data procedure that comments can be placed at the end of a line of code.

Expressions Reference

An expression enables you to combine variables, constants, operators, and functions in an operation that returns a single value. Expressions are used in a wide variety of Maintain Data commands. You can build increasingly complex expressions by combining simpler ones.

In this chapter:

- [Types of Expressions](#)
 - [Writing Numeric Expressions](#)
 - [Writing Date Expressions](#)
 - [Writing Date-Time Expressions](#)
 - [Writing Character Expressions](#)
 - [Writing Logical Expressions](#)
 - [Writing Conditional Expressions](#)
 - [Handling Null Values in Expressions](#)
-

Types of Expressions

This section describes the types of expressions that you can write in Maintain Data:

- Numeric.** Use a numeric expression to perform calculations that use numeric constants (integer or decimal) and variables. For example, you can write an expression to compute the bonus for each employee by multiplying the current salary by the desired percentage as follows:

```
COMPUTE Bonus = Curr_Sal * 0.05 ;
```

A numeric expression returns a numeric value. For details, see [Writing Numeric Expressions](#) on page 23.

- Date.** Use date expressions to perform numeric calculations on dates. For example, you can write an expression to determine when a customer can expect to receive an order by adding the number of days in transit to the date on which you shipped the order as follows:

```
COMPUTE Delivery/MDY = ShipDate + 5 ;
```

There are two types of date expressions:

- ❑ **Date expressions.** Return a date, a component of a date, or an integer that represents the number of days, months, quarters, or years between two dates. For details, see [Writing Date Expressions](#) on page 28.
- ❑ **Date-time expressions.** Use a variety of specialized date-time functions, each of which returns a different kind of value. For details, see [Writing Date-Time Expressions](#) on page 34.

- ❑ **Character.** Use a character expression to manipulate alphanumeric or text constants or variables. For example, you can write an expression to extract the first initial from an alphanumeric field as follows:

```
COMPUTE First_Init/Al = MASK (First_Name, '9$$$$$$$$') ;
```

A character expression returns a character value. For details, see [Writing Character Expressions](#) on page 39.

- ❑ **Logical.** Use a logical expression to evaluate the relationship between two values. A logical expression returns TRUE or FALSE. For details, see [Writing Logical Expressions](#) on page 44.
- ❑ **Conditional.** Use a conditional expression to assign a value based on the result of a logical expression. A conditional expression (IF...THEN...ELSE) returns a numeric or character value. For details, see [Writing Conditional Expressions](#) on page 46.

Reference: Usage Notes for Expressions

- ❑ Expressions in Maintain Data cannot exceed 40 lines of text or use more than 16 IF statements.
- ❑ Expressions are self-terminating. You do not use a semicolon to indicate the end of an expression. Semicolons are used only to terminate commands.

Expressions and Variable Formats

When you use an expression to assign a value to a variable, make sure that you give the variable a format that is consistent with the value returned by the expression. For example, if you use a character expression to concatenate a first name and last name and assign it to the variable FullName, make sure you define the variable as character (that is, as alphanumeric or text).

Writing Numeric Expressions

A numeric expression performs a calculation that uses numeric constants, variables, operators, or functions to return a numeric value. A numeric expression can consist of the following components, as highlighted below:

- ❑ A numeric constant. For example:

```
COMPUTE COUNT/I2 = 1 ;
```

- ❑ A numeric variable. For example:

```
COMPUTE RECOUNT/I2 = Count ;
```

- ❑ Two numeric constants or variables joined by a numeric operator. For example:

```
COMPUTE BONUS/D12.2 = CURR_SAL * 0.05 ;
```

- ❑ A numeric function. For example:

```
COMPUTE LONGEST_SIDE/D12.2 = MAX(WIDTH, HEIGHT) ;
```

- ❑ Two or more numeric expressions joined by a numeric operator. For example:

```
COMPUTE PROFIT/D12.2 = (RETAIL_PRICE - UNIT_COST) * UNIT_SOLD ;
```

Reference: Numeric Operators

The following list shows the numeric operators you can use in an expression:

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*

Operation	Operator
Division	/
Integer division	DIV
Remainder division	MOD
Exponentiation	**

Note: Multiplication, DIV, MOD, and exponentiation are not supported for date expressions of any type. To isolate part of a date, use a simple assignment command.

Syntax: **How to Use DIV: Integer Division**

The DIV operator can be used in any valid expression to perform integer division. The result is an integer value and the remainder is truncated.

The syntax is:

expression DIV expression

Example: **Using DIV to Perform Integer Division**

In this example, the DIV operator is used to calculate the number of whole days that are equivalent to a number of hours:

```
COMPUTE Days/I4 = Hours DIV 24;
```

Syntax: **How to Use MOD: Calculating the Remainder**

The MOD operator can be used in any valid expression to calculate the remainder when division is performed.

The syntax is:

expression MOD divisor

The MOD operator always returns an integer value, and all decimal places are truncated.

Example: **Using MOD to Calculate a Remainder**

In the following example, the divisor is 10. The variables IntMod and DblMod contain the result.


```

MAINTAIN FILE Car
FOR 4 NEXT Country MPG INTO StkCar
REPEAT StkCar.FocCount Cnt/I4=1;
    COMPUTE IntMod/I4=StkCar(Cnt).MPG MOD 10;
        DblMod/D4.1=StkCar(Cnt).MPG MOD 10;
    TYPE "MPG=<<StkCar(Cnt).MPG" " IntMod=<<IntMod    DblMod=<<DblMod"
ENDREPEAT Cnt=Cnt+1;
END

```

The decimal place in the variable DblMod is truncated, even though the format is D4.1.

```

MPG=    16 INTMOD=    6 DBLMOD=  6.0
MPG=    9 INTMOD=    9 DBLMOD=  9.0
MPG=   11 INTMOD=    1 DBLMOD=  1.0
MPG=   25 INTMOD=    5 DBLMOD=  5.0

```

Order of Evaluation

Maintain Data performs numeric operations in the following order:

1. Exponentiation.
2. Division and multiplication.
3. Addition and subtraction.

When operators are at the same level, they are evaluated from left to right. Because expressions in parentheses are evaluated before any other expression, you can use parentheses to change this predefined order. For example, the following expressions yield different results because of parentheses:

```

COMPUTE PROFIT = RETAIL_PRICE - UNIT_COST * UNIT_SOLD ;
COMPUTE PROFIT = (RETAIL_PRICE - UNIT_COST) * UNIT_SOLD ;

```

In the first expression, UNIT_SOLD is first multiplied by UNIT_COST, and the result is subtracted from RETAIL_PRICE. In the second expression, UNIT_COST is first subtracted from RETAIL_PRICE, and that result is multiplied by UNIT_SOLD.

Evaluating Numeric Expressions

Maintain Data follows a specific evaluation path for each numeric expression based on the format of the operands and the operators. If the operands all have the same format, most operations are carried out in that format. This is known as native-mode arithmetic. If the operands have different formats, Maintain Data converts the operands to a common format in a specific order of format precedence. Regardless of operand formats, some operators require conversion to specific formats so that all operands are in the appropriate format.

Identical Operand Formats

If all operands of a numeric operator are of the same format, you can use the following table to determine whether or not the operations are performed in that native format or if the operands are converted before and after executing the operation. In each case requiring conversion, operands are converted to the operational format and the intermediate result is returned in the operational format. If the format of the result differs from the format of the target variable, the result is converted to the format of the target variable.

Operation		Operational Format
Addition	+	Native.
Subtraction	-	Native.
Multiplication	*	Native.
Full Division	/	Accepts single-precision or double-precision floating point, converts all others to double-precision floating point.
Integer Division	<code>DIV</code>	Native, except converts packed decimal to double-precision floating point.
Remainder Division	<code>MOD</code>	Native, except converts packed decimal to double-precision floating point.
Exponentiation	**	Double-precision floating point.

Example: Identical Operand Formats

Because the following variables are defined as integers,

```
COMPUTE OperandOne/I4;
       OperandTwo/I4;
       Result/I4;
```

Maintain Data does the following multiplication in native-mode arithmetic (integer arithmetic):

```
COMPUTE Result = OperandOne * OperandTwo;
```

Different Operand Formats

If operands of a numeric operator have different formats, you can use the following table to determine what the common format is after Maintain Data converts them. Maintain Data converts the lower operand to the format of the higher operand before performing the operation.

Order	Format
1	16-byte packed decimal
2	Double-precision floating point
3	8-byte packed decimal
4	Single-precision floating point
5	Integer
6	Character (alphanumeric and text)

For example, if a 16-byte packed-decimal operand is used in an expression, all other operands are converted to 16-byte packed-decimal format for evaluation. However, if an expression includes only integer and alphanumeric operands, all alphanumeric operands are converted to integer format.

Maintain Data converts the alphanumeric to a numeric. If the alphanumeric is not a number, it is converted to 0 (zero), and 0 (zero) gets substituted into the equation.

If you assign a decimal value to an integer, Maintain Data truncates the fractional value.

Continental Decimal Notation

By default, you must use a decimal point (.) to indicate a decimal position when writing a value in a Maintain Data procedure (for example, a COMPUTE statement), and a comma (,) to demarcate thousands, regardless of the CDN setting.

To write the value in a procedure using the format matching the CDN setting for a value other than OFF (for example, ON, QUOTE, QUOTEP, SPACE), use MNTCON CDN_FEXINPUT ON in the EDASPROF file or user profile, and use quotation marks to delimit the value. You can use single quotation marks (') or double quotation marks (") when CDN=ON or SPACE. However, you must use double quotation marks (") when CDN=QUOTE or QUOTEP. The MNTCON CDN_FEXINPUT command does not apply to values entered in Maintain Data forms at run time.

For more information on setting MNTCON CDN_FEXINPUT, see [MNTCON CDN_FEXINPUT](#) on page 213.

When entering values in forms at run time, observe the following rule:

- If CDN=OFF or QUOTEP, use a decimal point (.) to enter decimal values.
- If CDN=ON, QUOTE or SPACE, use a comma (,) to enter decimal values.

For more information on the SET CDN command, see the *Developing Reporting Applications* manual.

Writing Date Expressions

A date expression returns a date, a component of a date, or an integer that represents the number of days, months, quarters, or years between two dates.

A date expression can consist of the following components, highlighted below:

- A date constant. For example:

```
COMPUTE StartDate/MDY= 'FEB 28 93';
```

Note the use of single quotation marks (') around the date constant FEB 28 93.

- A date variable. For example:

```
COMPUTE NewDate = StartDate;
```

- An alphanumeric, integer, or packed variable with date edit options. For example, in the second COMPUTE command, OldDate is a date expression:

```
COMPUTE OldDate/I6YMD = '980307';  
COMPUTE NewDate/YMD DFC 19 YRT 10 = OldDate;
```

- A calculation that uses addition, subtraction, or date functions to return a date. For example:

```
COMPUTE Delivery/MDY = ShipDate + 5;
```

- A calculation that uses subtraction or date functions to return an integer (not a date) that represents the number of days, months, quarters, or years between two dates. For example:

```
COMPUTE ResponseTime/I4 = ShipDate - OrderDate;
```

Formats for Date Values

Maintain Data enables you to work with dates in one of two ways:

- ❑ In date format, Maintain Data treats the value as a date. Date format interprets cross-century dates correctly, regardless of whether they are displayed with century digits. This is the preferred way of working with date values. The date is stored internally as an integer representing the number of days between the date and a standard base date. The base date is 12/31/1900 for all date variables declared in any operating environment using a 'D' for days, and also for all date variables declared in a Windows or UNIX environment using a 'Y' for years. The base date is 01/01/1901 for all date variables declared with a 'Y' in an S/390 environment.
- ❑ In integer, packed, or alphanumeric format with date edit options, Maintain Data treats the value as an integer, a packed decimal, or an alphanumeric string. When displaying the value, Maintain Data formats it to resemble a date.

You can convert a date in one format to a date in another format simply by assigning one to the other. For example, the following assignment statements take a date stored as an alphanumeric variable formatted with date edit options and convert it to a date stored as a date variable:

```
COMPUTE AlphaDate/A6MDY = '120599';
       RealDate/MDY = AlphaDate;
```

The following table illustrates how the format affects storage and display:

	Date Format For example: MDY		Integer, Packed, or Alphanumeric Format For example: A6MDY	
October 31, 1992	33542	10/31/92	103192	10/31/92
November 01, 1992	33543	11/01/92	110192	11/01/92

Evaluating Date Expressions

The format of a variable determines how you can use it in a date expression. Calculations on dates in date format can incorporate numeric operators, as well as numeric functions. If you need to perform calculations on dates in integer, packed, or alphanumeric format, we recommend that you first convert them to dates in date format, and then perform the calculations on the dates in date format.

Consider the following example, which calculates how many days it takes for your shipping department to fill an order by subtracting the date on which an item is ordered, OrderDate, from the date on which it is shipped, ShipDate:

```
COMPUTE TurnAround/I4 = ShipDate - OrderDate;
```

An item ordered on October 31, 1992 and shipped on November 1, 1992 should result in a difference of 1 day. The following table shows how the format affects the result:

	Value in Date Format	Value in Integer Format
ShipDate = November 1, 1992	33543	110192
OrderDate = October 31, 1992	33542	103192
TurnAround	1	7000

If the date variables are in integer format, you can convert them to date format and then calculate TurnAround:

```
COMPUTE NewShipDate/MDY = ShipDate;  
NewOrderDate/MDY = OrderDate;  
TurnAround/I4 = NewShipDate - NewOrderDate;
```

Selecting the Format of the Result Variable

A date expression always returns a number. That number may represent a date or the number of days, months, quarters, or years between two dates. When you use a date expression to assign a value to a variable, the format you give to the variable determines how the result is displayed.

Consider the following commands. The first command calculates how many days it takes for your shipping department to fill an order by subtracting the date on which an item is ordered, ORDERDATE, from the date on which it is shipped, SHIPDATE. The second calculates a delivery date by adding 5 days to the date on which the order is shipped, SHIPDATE.

```
COMPUTE TURNAROUND/I4 = SHIPDATE - ORDERDATE ;  
COMPUTE DELIVERY/MDY = SHIPDATE + 5 ;
```

In the first command, the date expression returns the number of days it takes to fill an order. Therefore, the associated variable, TURNAROUND, must have an integer format. In the second command, the date expression returns the date on which the item will be delivered. Therefore, the associated variable, DELIVERY, must have a date format.

Manipulating Dates in Date Format

This section provides additional information on how to write expressions using values represented in date format. It describes how to:

- ❑ Use a date constant in an expression.
- ❑ Extract a date component.
- ❑ Combine variables with different components in an expression.

Using a Date Constant in an Expression

When you use a date constant in a calculation with variables in date format, you must enclose it in single quotation marks ('), otherwise, Maintain Data interprets it as the number of days between the constant and the base date (December 31, 1900). The following example shows how to initialize STARTDATE with the date constant 02/28/93:

```
COMPUTE STARTDATE/MDY = '022893' ;
```

Extracting a Date Component

Date components include days, months, quarters, and years. You can write an expression that extracts a component from a variable in date format. The following example shows how you can extract a month from SHIPDATE, which has the format MDY:

```
COMPUTE SHIPMONTH/M = SHIPDATE ;
```

If SHIPDATE has the value November 23, 1992, the above expression returns the value 11 for SHIPMONTH. Note that calculations on date components automatically produce a valid value for the desired component. For example, if the current value of SHIPMONTH is 11, the following expression correctly returns the value 2, not 14:

```
COMPUTE ADDTHREE/M = SHIPMONTH + 3 ;
```

You cannot write an expression that extracts days, months, or quarters from a date that did not have these components. For example, you cannot extract a month from a date in YY format, which represents only the number of years.

Combining Variables With Different Components in an Expression

When using variables in date format, you can combine variables with a different order of components within the same expression. For example, consider the following two variables, where DATE_PAID has the format YYMD and DUE_DATE has the format MDY. You can combine these two variables in an expression to calculate the number of days that a payment is late, such as the following expression:

```
COMPUTE DAYS_LATE/I4 = DATE_PAID - DUE_DATE ;
```

In addition, you can assign the result of a date expression to a variable with a different order of components from the variables in the expression. For example, consider the variable DATE_SOLD, which contains the date on which an item is sold, in YYMD format. You can write an expression that adds 7 days to DATE_SOLD to determine the last date on which the item can be returned, and then assign the result to a variable with DMY format, as in the following COMPUTE command:

```
COMPUTE RETURN_BY/DMY = DATE_SOLD + 7 ;
```

Different Operand Date Formats

In an expression in a procedure, all date formats are valid. If you have an expression that operates on date variables with different formats (for example, QY and MDY), Maintain Data converts one variable to the format of the other variable in order to perform the operation.

However, there are a few types of date variables that you cannot use in a mixed-format date expression. These variables, formatted as single components, such as a day of the week or year (formats D, W, Y, and YY), cannot be meaningfully converted to a more complete date (such as a year with a month). You can use these date variables in same-type date expressions.

If a date with format M is compared to a date with format Q (or vice versa), the operand on the right is converted to the format of the operand on the left, and then the comparison is performed.

For all other date-to-date comparisons, the date with the lesser format is promoted to the format of the higher date, where possible. If conversion is not possible, an error is generated.

The following conversion hierarchy applies to date formats:

Order	Date Format
1	Dates with three components (for example, MDY, YYMD, Julian dates).
2	Dates with two components, one of which is a month (for example, MYY or YM).
3	Dates with two components, one of which is a quarter (for example, YQ).
4	Single component M or Q.
5	All other formats.

Dates in the fifth category do not get promoted.

When you have dates of two different types, dates in the lower category are promoted to the higher type.

Using Addition and Subtraction in a Date Expression

When performing addition or subtraction in a date expression:

- ❑ Adding a number to a date yields a date. It is up to you to make sure that the expression resolves to a meaningful value.
- ❑ Subtracting one date from another yields an integer that represents the difference between the two dates.
- ❑ When a date with format M or Q is subtracted from a higher type of date, the operand on the right is converted to the format of the operand on the left.
- ❑ When a two-component date is subtracted from a three-component date, or vice versa, the variable with the lesser format is promoted to the type of the variable with the higher format.
- ❑ When subtracting a Q format date from an M format date, or vice versa, the operand on the right is converted to the same format as the operand on the left.
- ❑ Subtracting a number from a date yields a date with the same format as the original date.
- ❑ You cannot subtract a date from a number, and you cannot add a date to a date.

Example: Using Addition and Subtraction in a Date Expression

Given the following variable definitions

```
DECLARE Days/D = 23;
DECLARE OldYear/YY = 1960;
DECLARE NewYear/YY = 1994;
DECLARE YearsApart/YY;
DECLARE OldYearMonth/YM = 9012;
DECLARE NewYearMonth/YM;
DECLARE FullDate/YMD = 870615;
```

the following COMPUTE commands are valid:

```
COMPUTE
YearsApart = NewYear - OldYear;
NewYear = OldYear + 2;
NewYearMonth = OldYearMonth - FullDate;
```

However, the next series of COMPUTE commands are invalid, because they include date variables formatted as just a day (Days) or just a year (OldYear) in a mixed-format date expression:

```
COMPUTE  
NewYear = FullDate - OldYear;  
FullDate = OldYearMonth + Days;
```

Writing Date-Time Expressions

Date-time values for Maintain Data may be supplied in one of the following ways:

- As a value in a computed expression, enclosed in single quotation marks (') or double quotation marks (").
- As a value extracted or computed by a date-time function.
- Using an application Winform.

Maintain Data supports the date-time data type with the following restrictions:

- The default date-time format separators (/) must be used. Other separators are not supported.
- When you create a WHERE statement or an IF THEN ELSE clause, you must use a variable as the test value.
- The format SET DATEFORMAT, used to change the default input format, is not supported.
- The SET commands WEEKFIRST and DTSTRICT are not supported.
- Computing an expression to DT (value) is not supported.

Syntax: How to Write Date-Time Expressions

A date-time constant in a Maintain Data procedure, and in an IF expression in a report procedure, has one of the following formats.

Note: In an IF expression, if the value contains no blanks or special characters, the single quotation marks (') are not necessary.

```
'date_string [time_string]' 'time_string [date_string]'
```

where:

time_string

Cannot contain blanks. Time components are separated by colons and may be followed by AM, PM, am, or pm. For example:

```

14:30:20:99      (99 milliseconds)
14:30
14:30:20.99     (99/100 seconds)
14:30:20.999999 (999999 microseconds)
02:30:20:500pm

```

Note that seconds can be expressed with a decimal point or be followed by a colon.

- ❑ If there is a colon after seconds, the value following it represents milliseconds. There is no way to express microseconds using this notation.
- ❑ A decimal point in the seconds value indicates the decimal fraction of a second. Microseconds can be represented using six decimal digits.

date_string

Can have one of the following three formats:

- ❑ **Numeric string format.** Is exactly four, six, or eight digits. Four-digit strings are considered to be a year (century must be specified). The month and day are set to January 1. Six-digit and eight-digit strings contain two or four digits for the year, followed by two for the month, and then two for the day.

If a numeric-string format longer than eight digits is encountered, it is treated as a combined date-time string in the *Hn* format described in the *Describing Data With WebFOCUS Language* manual. The following are examples of numeric string date constants:

```

99
1999
19990201

```

- ❑ **Formatted-string format.** Contains a one-digit or two-digit day, a one-digit or two-digit month, and a two-digit or four-digit year separated by spaces, slashes, hyphens, or periods. If any of the three fields is four digits, it is interpreted as the year, and the other two fields must follow the order given by the DATEFORMAT setting. The following are examples of formatted-string date constants:

```

1999/05/20
5 20 1999
99.05.20
1999-05-20

```

- ❑ **Translated-string format.** Contains the full or abbreviated month name. The year must also be present in four-digit or two-digit form. If the day is missing, day 1 of the month is assumed. If the day is present, it can have one or two digits. If the string contains both a two-digit year and a two-digit day, they must be in the order given by the DATEFORMAT setting. For example:

January 6 2000

Note:

- ❑ The date and time strings must be separated by at least one blank space. Blank spaces are also permitted at the beginning and end of the date-time string.
- ❑ In each date format, two-digit years are interpreted using the [F]DEFCENT and [F]YRTHRESH settings.

Example: Using a Date-Time Value in a COMPUTE Command

```
COMPUTE RAISETIME/HYYMDIA = '20000101 09:00AM';
```

Manipulating Date-Time Values Directly

The only direct operations that can be performed on date-time variables and constants are comparison using a logical expression and simple assignment of the form $A = B$. All other operations are accomplished through a set of date-time subroutines. For more information, see [Writing Character Expressions](#) on page 39.

Comparing and Assigning Date-Time Values

Any two date-time values can be compared, even if their lengths do not match.

If a date-time field supports missing values, fields that contain the missing value have a greater value than any date-time field can have. Therefore, in order to exclude missing values from report output when using a GT or GE operator in a selection test, it is recommended that you add the additional constraint *field* NE MISSING to the selection test:

```
date_time_field {GT|GE} date_time_value AND date_time_field NE MISSING
```

Assignments are permitted between date-time formats of equal or different lengths. Assigning a 10-byte date-time value to an 8-byte date-time value truncates the microsecond portion (no rounding takes place). Assigning a short value to a long one sets the low-order three digits of the microseconds to zero.

Other operations, including arithmetic, concatenation, and the reporting operators EDIT and LIKE on date-time operands are not supported. Reporting prefix operators that work with alphanumeric fields are supported.

Example: Testing for Missing Date-Time Values

Consider the DATETIM2 Master File:

```
FILE=DATETIM2, SUFFIX=FOC , $
SEGNAME=DATETIME, SEGTYPE=S0 , $
FIELD=ID, ID, USAGE = I2 , $
FIELD=DT1, DT1, USAGE=HYMDS, MISSING=ON, $
```

Field DT1 supports missing values. Consider the following request:

```
TABLE FILE DATETIM2
PRINT ID DT1
END
```

The resulting report output shows that in the instance with ID=3, the field DT1 has a missing value:

```
ID  DT1
--  ---
 1  2000/01/01 02:57:25
 2  1999/12/31 00:00:00
 3  .
```

The following request selects values of DT1 that are greater than 2000/01/01 00:00:00 and are not missing:

```
TABLE FILE DATETIM2
PRINT ID DT1
WHERE DT1 NE MISSING AND DT1 GT DT(2000/01/01 00:00:00);
END
```

The missing value is not included in the report output:

```
ID  DT1
--  ---
 1  2000/01/01 02:57:25
```

Date-Time Subroutines

The following subroutines allow you to manipulate date-time values:

Function Name	Description
HADD	Increments date-time values by a specified number of units.
HCVRT	Converts date-time values to alphanumeric format for use with operators, such as EDIT, CONTAINS, and LIKE.
HDATE	Extracts the date components from a date-time field and converts them to a date field.

Function Name	Description
HDIFF	Returns the number of units of a specific date-time component between two date-time values.
HDTTM	Converts a date field to a date-time field with the time set to midnight.
HEXTR	Extracts components from a date-time value and moves them to a target date-time field with all other components set to zero.
HGETC	Returns the current date and time in date-time format.
HMASK	Extracts components from a date-time value and moves them to a target date-time field with all other components of the target field preserved.
HHMMSS	Retrieves the current time from the system.
HINPUT	Converts an alphanumeric string to a date-time value.
HMIDNT	Changes the time portion of a date-time field to midnight.
HNAME	Extracts specified components of a date-time value and converts them to alphanumeric format.
HPART	Extracts a component of a date-time value in numeric format.
HSETPT	Inserts the numeric value of a specified component in a date-time field.
HTIME	Extracts all of the time components from a date-time field and converts them to a number of milliseconds or microseconds in numeric format.
HTMTOTS/TIMETOTS	Converts a time to a timestamp.

For more information on these functions, see the *Using Functions* manual.

Reference: Notes Regarding ISO Standard Date-Time Representations

International Standard ISO 8601 describes the standards for numeric representations of date and time. Some of the relevant standards and notes about their implementation follow:

- ❑ The international standard date notation is YYYY-MM-DD.
- ❑ The international standard for the first day of a week is Monday. You can use the WEEKFIRST parameter with App Studio procedures to control the day used as the first day of the week by the date-time functions. However, Maintain Data does not support this setting.
- ❑ The standard specifies that week 1 of a year is the first week of the year that has a Thursday. Combined with the standard of Monday as day 1, this rule ensures that week 1 has at least four of its days in the specified year.

The following rules represent an extension to the standard in this implementation:

- ❑ Whatever day you choose for your WEEKFIRST setting, the date-time functions define week 1 as the first week with at least four days in the specified year.
- ❑ With these rules, it is possible for the first few days of January to fall in the week prior to week 1. The international standard considers these dates to be in week 53 of the previous year. However, the date-time functions return zero for the week component when it falls in the week prior to week 1.
- ❑ The international standard notation for the time of day is hh:mm:ss using the 24-hour system. However, the date-time data type and date-time functions allow you to use the 12-hour system.

Writing Character Expressions

A character expression returns an alphanumeric or text value.

A character expression can consist of the following components, highlighted below:

- ❑ An alphanumeric or text constant (that is, a character string enclosed in single quotation marks ('') or double quotation marks ("")). For example:

```
COMPUTE STATE = 'NY' ;
```

- ❑ An alphanumeric or text variable. For example:

```
COMPUTE AddressPartTwo = STATE ;
```

- ❑ A function returning an alphanumeric or text result. For example:

```
COMPUTE INITIAL/A1= MASK(FIRSTNAME, '9$$$$$$$$');
```

- ❑ Two or more alphanumeric and/or text expressions combined into a single expression using the concatenation operator. For example:

```
COMPUTE TITLE/A19= 'DR.' || LAST_NAME;
```

Concatenating Character Strings

You can write an expression to concatenate several alphanumeric and/or text values into a single character string. The concatenation operator takes one of two forms, as shown in the following table:

Symbol	Represents	Function
	Weak concatenation.	Preserves trailing spaces.
	Strong concatenation.	Suppresses trailing spaces.

Evaluating Character Expressions

Any non-character expression that is embedded in a character expression is automatically converted to a character string.

A constant must be enclosed in single quotation marks (') or double quotation marks ("). Whichever delimiter you choose, you must use the same one to begin and end the string. The ability to use either single quotation marks (') or double quotation marks (") provides the added flexibility of being able to use one kind of quotation mark to enclose the string, and the other kind as data within the string itself.

The backslash (\) is the escape character. You can use it to:

- ❑ Include a delimiter of a string (for example, a single quotation mark (')) within the string itself, as part of the value. Simply precede the character with a backslash (\), and Maintain Data will interpret the character as data, not as the end-of-string delimiter.
- ❑ Include a backslash within the string itself, as part of the value. Simply precede the backslash with a second backslash (\\).
- ❑ Generate a line feed (for example, when writing a message to a file or device using the SAY command). Simply follow the backslash with the letter n (\n).

When the backslash is used as an escape character, it is not included in the length of the string.

Example: **Using Quotation Marks in a Character Expression**

Because you can define a character string using single quotation marks (') or double quotation marks ("), you can use one kind of quotation mark to define the string and the other kind within the string, as in the following expressions:

```
COMPUTE LastName = "O'HARA";
COMPUTE Msg/A40 = 'This is a "Message"';
```

Example: **Using a Backslash Character (\) in a Character Expression**

You can include a backslash (the escape character) within a string as part of the value by preceding it with a second backslash. For example, the following source code

```
COMPUTE Line/A40 = 'The characters \\\" are interpreted as \';
.
.
.
TYPE "Escape info: <Line"
```

displays:

```
Escape info: The characters \ are interpreted as '
```

When the backslash is used as an escape character, it is not included in the length of the string. For example, a string of five characters and one escape character fits into a five-character variable:

```
COMPUTE Word/A5 = 'Can\t'
```

Example: **Specifying a Path in a Character Expression**

A path may, depending on the operating system, contain backslashes (\). Because the backslash is the escape character for character expressions, if you specify a path that contains backslashes in an expression, you must precede each of the backslashes with a second backslash. For example:

```
MyLogo/A50 = "C:\\\ibi_img\\AcmeLogo.gif";
```

Example: **Extracting Substrings and Using Strong and Weak Concatenation**

The following example shows how to use the SUBSTR function to extract the first initial from a first name, and then use both strong and weak concatenation to produce the last name, followed by a comma (,), followed by the first initial, followed by a period:

```
First_Init/A1 = SUBSTR (First_Name, 1, 1);  
Name/A19 = Last_Name || (' , ' | First_Init | '.');
```

Suppose that `First_Name` has the value `Chris` and `Last_Name` has the value `Edwards`. The above request evaluates the expressions as follows:

1. The `SUBSTR` function extracts the initial `C` from `First_Name`.
2. The expression in parentheses is evaluated. It returns the value

```
, C.
```

3. `Last_Name` is concatenated to the string derived in step 2 to produce the following:

```
Edwards, C.
```

Note that while `Last_Name` has the format `A15`, strong concatenation suppresses the trailing spaces.

Variable-Length Character Variables

You can enable a character variable to have a varying length by declaring it either as text (`TX`) or as alphanumeric with a length of zero (`AO`). `TX` and `AO` are equivalent.

Specifying a varying length provides several advantages:

- ❑ **Increased length.** The variable can be as long as 32,766 characters. A fixed-length character variable, by contrast, has a maximum of 256 characters.
- ❑ **Flexible logic.** Variable length enables you to declare one variable that can accept values of many different lengths (ranging from zero to 32,766 characters). Other alphanumeric variables, by contrast, are always of fixed length.

The default value of a variable-length character variable is a string of length zero.

- ❑ **No padding.** If you assign a character string to a longer fixed-length alphanumeric variable, the variable pads the value of the string with spaces to make up the difference. If you assign the same string to a variable-length variable, it stores the original value without padding it with spaces.

Of course, if you assign a string with trailing spaces to either a fixed-length or variable-length character variable, the variable preserves those trailing spaces.

- ❑ **Optimized memory usage.** The memory used by a variable-length character variable is proportional to its size. The shorter the value, the less memory is used.

Note that the characteristics of variable-length data source fields differ from those of temporary variables. When declaring a data source field, TX is supported for relational data sources, and has a maximum length of 4094 characters. A0 is not supported for data source fields. For information about data source text fields in Maintain Data applications, see the *Describing Data With WebFOCUS Language* manual.

Example: Padding and Trailing Spaces in Character Variables

Variable-length character variables, unlike those of fixed length, never pad strings with spaces.

For example, if you assign a string of 11 characters to a 15-character fixed-length alphanumeric variable, the variable pads the value with four spaces to make up the difference.

For example, the following source code

```
COMPUTE Name/A15 = 'Fred Harvey' ;
TYPE "<<Name End of string" ;
```

displays:

```
Fred Harvey    End of string
```

If you assign the same string of 11 characters to a variable-length variable, the variable stores the original value without padding it. For example, the following source code, in which Name is changed to be of variable length (specified by A0)

```
COMPUTE Name/A0 = 'Fred Harvey' ;
TYPE "<<Name End of string" ;
```

displays:

```
Fred HarveyEnd of string
```

If you assign a string with trailing spaces to a variable (of either fixed or varying length), the variable preserves those spaces. For example, the following source code

```
COMPUTE Name/A0 = 'Fred Harvey   ' ;
TYPE "<<Name End of string" ;
```

displays:

```
Fred Harvey   End of string
```

Writing Logical Expressions

A logical expression determines whether a particular condition is true. There are two kinds of logical expressions, relational and Boolean. The entities to compare determine the kind of expression.

- ❑ A relational expression returns TRUE or FALSE based on comparison of two individual values (either variables or constants).
- ❑ A Boolean expression returns TRUE or FALSE based on the outcome of two or more relational expressions.

You can use a logical expression to assign a value to a numeric variable. If the expression is true, the variable receives the value 1. If the expression is false, the variable receives the value 0.

Relational Expressions

A relational expression returns TRUE or FALSE based on the comparison of two individual values (either variables or constants). The following syntax lists the operators you can use in a relational expression:

character_expression char_operator character_constant

numeric_expression numeric_operator numeric_constant

where:

char_operator

Can be any of the following: EQ, NE, OMITS, CONTAINS.

numeric_operator

Can be any of the following: EQ, NE, LE, LT, GE, GT.

Boolean Expressions

Boolean expressions return a value of true (1) or false (0) based on the outcome of two or more relational expressions. Boolean expressions are often used in conditional expressions, which are described in [Writing Conditional Expressions](#) on page 46. You can also assign the result of a Boolean expression to a numeric or character variable, which will be set to 1 (if the expression is true) or 0 (if it is false). They are constructed using variables and constants connected by operators.

Syntax: How to Use Boolean Expressions

The syntax of a Boolean expression is:

```
(relational_expression) {AND|OR} (relational_expression)
NOT (logical_expression)
```

Boolean expressions can themselves be used as building blocks for more complex expressions. Use AND or OR to connect the expressions and enclose each expression in parentheses.

Evaluating Logical Expressions

If you assign a Boolean expression to a character variable, it may have the values TRUE, FALSE, 1, or 0. TRUE and 1 are equivalent, as are FALSE and 0. A numeric variable may have the values 1 or 0.

Alphanumeric constants with embedded blanks used in the expression must be enclosed in single quotation marks ('). An example is:

```
IF NAME EQ 'JOHN DOE'
```

OR cannot be used between constants in a relational expression. For example, the following expression is not valid

```
IF COUNTRY EQ 'US' OR 'BRAZIL' OR 'GERMANY'
```

Instead, it should be coded as a sequence of relational expressions:

```
IF (COUNTRY EQ 'US') OR (COUNTRY EQ 'BRAZIL') OR (COUNTRY EQ 'GERMANY')
```

Reference: Logical Operators

The following list shows the logical operators you can use in an expression:

Description	Operator
Equality	EQ
Inequality	NE
Less than	LT
Greater than	GT
Less than or equal to	LE
Greater than or equal to	GE

Description	Operator
Contains the specified character string	CONTAINS
Omits the specified character string	OMITS
Negation	NOT
Conjunction	AND
Disjunction	OR

Boolean operators are evaluated after numeric operators from left to right in the following order of priority:

Order	Operators
1	EQ NE LE LT GE GT NOT CONTAINS OMITS
2	AND
3	OR

Writing Conditional Expressions

A conditional expression assigns a value based on the result of a logical expression. The assigned value can be numeric or character.

Syntax: How to Use Conditional Expressions

The syntax of a conditional expression is

```
IF boolean THEN {expression1} [ELSE {expression2}]
```

where:

boolean

Is a Boolean expression. Boolean expressions are described in [Boolean Expressions](#) on page 44.

expression

Is a numeric, character, date, or conditional expression.

When the Boolean expression is true, the conditional expression returns the THEN expression. Otherwise, it returns the ELSE expression if one is provided.

The THEN and ELSE expressions can themselves be conditional expressions. If the expression following THEN is conditional, it must be enclosed in parentheses. A conditional expression can have up to 16 IF statements.

The variable to which you assign the conditional expression must have a format compatible with the formats of the THEN and ELSE expressions.

Handling Null Values in Expressions

When data does not exist for a variable, Maintain Data assigns the following default value, depending on how the format of the variable has been defined:

Data Type	Default value without the MISSING attribute	Default value with the MISSING attribute
Numeric	zero (0)	null
Date and time	space	null
Character	space	null

A null value (sometimes known as missing data) appears as a period (.) by default. You can change the character representation of the null value by issuing the SET NODATA command. For details, see the *Developing Reporting Applications* manual.

Null values affect the results of expressions that perform aggregating calculations such as averaging and summing. For information about the MISSING attribute in Master Files and the effect of null values in calculations, see the topics about null data and missing data in [Assigning Null Values: The MISSING Constant](#) on page 47.

Assigning Null Values: The MISSING Constant

You can assign the MISSING constant (that is, the null value) to variables (data source fields and temporary variables) that were defined with the MISSING attribute.

When you create a user-defined variable with the MISSING attribute and do not explicitly assign a value, it is created with the null value. For example, in the following command, Name is created with a null value:

```
COMPUTE Name/A15 MISSING ON = ;
```

Syntax: **How to Assign Null Values: The MISSING Constant**

The syntax for assigning a null value to an existing variable is:

```
COMPUTE target_variable = MISSING;
```

Example: **Assigning Null Values**

Suppose that the variable AcctBalance had been defined with the MISSING attribute. The command below assigns the null value to AcctBalance:

```
COMPUTE AcctBalance = MISSING;
```

Conversion in Mixed-Format Null Expressions

When a variable with a null value is assigned to a variable that is not defined with the MISSING attribute, the null value is converted to a zero or a space. For example, when the variable Q is assigned to R, the null value from Q is converted to a zero, because zero is the default value for numeric variables without the MISSING attribute.

```
Q/I4 MISSING ON = MISSING;  
R/I4 = Q;
```

The same conversion occurs before any mathematical operations are applied if the variables are used as operands in arithmetic expressions.

Testing Null Values

You may test for the null value using comparison operators EQ or NE in an expression. You can test any variable that has been declared with the MISSING attribute. The null value is represented by the MISSING constant.

Syntax: **How to Test Null Values**

The syntax for testing whether a value is null is:

```
target_variable {EQ|NE} MISSING
```

Example: **Testing Null Values**

In this example, an IF command executes a BEGIN block if the variable Returns is null:


```
IF Returns EQ MISSING THEN BEGIN  
.  
.  
.  
ENDBEGIN
```


Chapter 3

Command Reference

This topic provides a summary of the Maintain Data language commands and system variables, grouped by primary use. It also describes some commands that are outside the language but can be used to manage Maintain Data procedures. It then describes each command and system variable in detail.

When you develop an application, you can generate Maintain Data commands by:

- Using the Language Wizard in the Maintain Data Editor. The Wizard asks you questions about the logic you need to create, and automatically generates the required commands.
- Coding the commands yourself in the Maintain Data Editor.

Language Summary

This topic summarizes all Maintain Data language commands, grouping them by their primary use, such as transferring control or selecting records. Each command and system variable is described in detail later in this chapter.

Defining a Procedure

The basic syntax consists of the commands that start and terminate a Maintain Data procedure. The commands are:

`MAINTAIN`

Initiates the parsing and execution of a Maintain Data procedure. It is always the first line of the procedure.

`END`

Terminates the execution of a Maintain Data procedure.

Defining a Maintain Data Function (a Case)

The following command defines Maintain Data functions:

`CASE`

Defines a Maintain Data function. Maintain Data functions are also known as cases.

Defining Blocks of Code

The following command defines a block a code:

BEGIN

Defines a group of commands as a single block and enables you to issue them as a group. You can place a BEGIN block anywhere individual commands can appear.

Transferring Control

You can transfer control to another function within the current procedure, as well as to another procedure.

The commands that transfer control are:

PERFORM

Transfers control to another function. When the function finishes, control is returned to the command following PERFORM. You can also call a function directly, without PERFORM.

GOTO

Transfers control to another function or to a special label within the current function. When the function finishes, control does not return. You can also call a function directly, without GOTO.

CALL

Executes another Maintain Data procedure.

EXEC

Executes an external (non-Maintain Data) procedure.

Executing Procedures

The following commands run procedures, or prepare them for execution:

CALL

Executes a Maintain Data procedure, and enables you to pass data from the calling procedure.

EXEC

Executes an App Studio procedure.

Using Loops

The following command supports looping:

`REPEAT`

Enables a circular flow of control.

Using Forms

The following command is responsible for presentation logic:

`Winform`

Displays a form by which end users can read, enter, and edit data.

Defining Classes

The following command enables you to define classes:

`DESCRIBE`

Defines classes and data type synonyms.

Creating Variables

The following commands enable you to create variables:

`DECLARE`

Creates local and global variables, including objects.

`COMPUTE`

Creates global variables, including global objects. It can also assign values to existing variables.

Assigning Values

Maintain Data enables you to assign values to existing variables using the following command:

`COMPUTE`

Assigns values to existing variables.

Manipulating Stacks

Maintain Data provides several stack commands to manage the contents of stacks. Unless otherwise specified, each command operates on all rows in the stack. The following example copies the contents of the Indata stack to the Outdata stack:

```
FOR ALL COPY FROM Indata INTO Outdata;
```

One row or a range of rows may be specified to limit which rows are affected. For example, the following copies 100 records of the Indata stack, starting from the fourth record, and places them into the Outdata stack.

```
FOR 100 COPY FROM Indata(4) INTO Outdata;
```

The stack commands are:

COPY

Copies data from one stack to another.

STACK SORT

Sorts data in a stack.

STACK CLEAR

Initializes a stack.

INFER

Defines the columns in a stack.

In addition, there are two variables associated with a stack which can be used to manipulate individual rows or groups of rows in the stack. The stack variables are:

FocCount

Is the number of rows in the stack.

FocIndex

Is a pointer to the current instance in the stack.

Selecting and Reading Records

The record selection commands retrieve data from the data source and change position in the data source.

The commands are:

NEXT

Starts at the current position and moves forward through the data source. NEXT can retrieve data from one or more rows.

MATCH

Searches the entire segment for a matching field value. It retrieves an exact match in the data source.

REPOSITION

Changes the data source position to be at the beginning of the chain.

In addition, there is a system variable that provides a return code for NEXT and MATCH:

FocFetch

Signals the success or failure of a NEXT or MATCH command.

You can use the following commands to directly interface with a DBMS:

SYS_MGR.PRE_MATCH

Turns off preliminary database operation checking before an update.

SYS_MGR.GET_PRE_MATCH

Determines whether prematch checking is on or off.

SYS_MGR.ENGINE

Passes SQL commands directly to a DBMS.

SYS_MGR.DBMS_ERRORCODE

Retrieves a DBMS return code after an operation.

Conditional Actions

The conditional commands are:

IF

Issues a command depending on how an expression is evaluated.

ON MATCH

Determines the action to take when the prior MATCH command succeeds.

ON NOMATCH

Defines the action to take if the prior MATCH fails.

ON NEXT

Defines the action to take if the prior NEXT command succeeds.

ON NONEXT

Defines the action to take if the prior NEXT command fails.

Writing Transactions

The commands that can be used to control transactions are:

INCLUDE

Adds one or more new data source records.

UPDATE

Updates the specified data source fields or columns. Can update one or more records at a time.

REVISE

Adds new records to the data source and updates existing records.

DELETE

Deletes one or more records from the data source.

COMMIT

Makes all data source changes since the last COMMIT permanent.

ROLLBACK

Cancels all data source changes made since the last COMMIT.

There are several system variables that you can use to determine the success or failure of a data source operation or an entire logical transaction:

FocCurrent

Signals the success or failure of a COMMIT or ROLLBACK command.

FocError

Signals the success or failure of an INCLUDE, UPDATE, REVISE, or DELETE command.

FocErrorRow

If an INCLUDE, UPDATE, REVISE, or DELETE command that writes from a stack fails, this returns the number of the row that caused the error.

You can use the following commands to directly interface with a DBMS:

SYS_MGR.PRE_MATCH

Turns off preliminary database operation checking before an update.

SYS_MGR.GET_PRE_MATCH

Determines whether prematch checking is on or off.

SYS_MGR.ENGINE

Passes SQL commands directly to a DBMS.

SYS_MGR.DBMS_ERRORCODE

Retrieves a DBMS return code after an operation.

Setting Reporting Server Parameters

You can communicate with the Reporting Server using the following commands:

SET

Sets Reporting Server parameters. This command is outside the Maintain Data language, but is described in this chapter for your convenience.

SYS_MGR.FOCSET

Sets Reporting Server parameters, without having to set them in EDASPROF.

Using Libraries of Classes and Functions

You can import libraries using the following command:

MODULE

Imports a library of shared class definitions or functions into a Maintain Data procedure.

Messages and Logs

You can write messages to files, consoles, and forms using the following commands:

SAY

Writes messages to a file or to the default output device.

TYPE

Writes messages to a file or a form.

In addition, there is a system stack that is automatically populated with messages posted to the default output device by Maintain Data procedures, except for the starting procedure and external procedures:

FocMsg

Contains messages posted by Maintain Data and App Studio procedures.

BEGIN

The BEGIN/ENDBEGIN construction enables you to issue a set of commands. Because you can use this construction anywhere an individual Maintain Data command can be used, you can use a set of commands where before you could issue only one command. For example, it can follow ON MATCH, ON NOMATCH, ON NEXT, ON NONEXT, or IF.

Syntax: How to Use the BEGIN Command

The syntax for the BEGIN command is

```
BEGIN
  command
  .
  .
  .
ENDBEGIN
```

where:

BEGIN

Specifies the start of a BEGIN/ENDBEGIN block.

Note: You cannot assign a label to a BEGIN/ENDBEGIN block of code or execute it outside the bounds of the BEGIN/ENDBEGIN construction in a procedure.

command

Is one or more Maintain Data commands, except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, and MODULE. BEGIN blocks can be nested, allowing you to place BEGIN and ENDBEGIN commands between BEGIN and ENDBEGIN commands.

ENDBEGIN

Specifies the end of a BEGIN block.

Example: BEGIN With ON MATCH

The following example illustrates a block of code that executes when MATCH is successful:

```
MATCH Emp_ID
  ON MATCH BEGIN
    COMPUTE Curr_Sal = Curr_Sal * 1.05;
    UPDATE Curr_Sal;
    COMMIT;
  ENDBEGIN
```

Example: BEGIN With ON NEXT

The following example shows BEGIN and ENDBEGIN with ON NEXT:

```
ON NEXT BEGIN
  TYPE "Next successful.";
  COMPUTE New_Sal = Curr_Sal * 1.05;
  PERFORM Cleanup;
ENDBEGIN
```

Example: **BEGIN With IF**

You can also use BEGIN and ENDBEGIN with IF to run a set of commands depending on how an expression is evaluated. In the following example, BEGIN and ENDBEGIN are used with IF and FocError to run a series of commands when the prior command fails:

```
IF FocError NE 0 THEN BEGIN
  TYPE "There was a problem.";
  .
  .
  .
ENDBEGIN
```

Example: **Nested BEGIN Blocks**

The following example nests two BEGIN blocks. The first block starts if there is a MATCH on Emp_ID and the second block starts if UPDATE fails:

```
MATCH Emp_ID FROM Emps(Cnt);
ON MATCH BEGIN
  TYPE "Found employee ID <Emps(Cnt).Emp_ID";
  UPDATE Department Curr_Sal Curr_JobCode Ed_Hrs
    FROM Emps(Cnt);
  IF FocError GT 0 THEN BEGIN
    TYPE "Was not able to update the data source.";
    PERFORM Errorhnd;
  ENDBEGIN
ENDBEGIN
```

CALL

Use the CALL command when you need one procedure to call another. When you use CALL, both the calling and called procedures communicate using variables. Local variables that you pass between them and the global transaction variables FocError, FocErrorRow, and FocCurrent. CALL allows you to link modular procedures, so each procedure can perform its own set of discrete operations within the context of your application. Since called procedures can reside on different servers, you can physically partition applications across different platforms.

Syntax: **How to Use the CALL Command**

The syntax of the CALL command is:

```
CALL procedure [KEEP|DROP] [PATH {VAR|LIST}] [FROM var_list]  
  [INTO var_list] [;]  
  var_list: {variable} [{variable} ...]
```

where:

procedure

Is the name of the Maintain Data procedure to run.

KEEP | DROP

The DROP parameter terminates the server session. The KEEP parameter leaves the server session active for reuse by subsequent calls. KEEP is the default value.

PATH

Is used to specify additional locations (search paths) the system should use when searching for dependent resources (Master Files, imported modules, and others). The path location names are application names existing within the APPROOT directory structure or application names that have been introduced with the APP MAP command. The search path value can be in the form of a Maintain Data variable or a list of literal values enclosed in double quotation marks ("), as follows:

```
CALL Procedure PATH "AppDir1 AppDir2 AppDir3" ;  
CALL Procedure PATH MyVariable ;
```

FROM

Is included if the Maintain Data procedure passes one or more variables to the called procedure.

INTO

Is included if the called Maintain Data procedure passes one or more variables back to this procedure.

var_list

Are the scalar variables and stacks that are passed to or from this procedure. Multiple variables are separated by blank spaces.

variable

Is the name of a scalar variable or stack. You can pass any variable except for those defined as variable-length character (that is, those defined as AO or TX) and those defined using STACK OF.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Calling Procedures to Validate Data

The following example shows three Maintain Data procedures. The first displays a form to collect employee IDs and salaries. It then calls Validate to make sure that the salaries are in a range. If they are all valid, it calls PutData and includes them in the data source. If not, it sets FocError to the invalid row and redisplay the data.

```

MAINTAIN FILE EMPLOYEE
INFER EMP_ID CURR_SAL INTO EMPSTACK;
Winform Show EMPL;

CASE VALIDATE_DATA
CALL VALIDATE FROM EMPSTACK;
IF FOCERROR EQ 0 THEN BEGIN
    CALL PUTDATA FROM EMPSTACK;
    TYPE "DATA ACCEPTED";
ENDBEGIN

ELSE BEGIN
    TYPE "THERE WAS AN ERROR IN ROW <FOCERROR";
    TYPE "TRY AGAIN";
ENDBEGIN
ENDCASE
END

```

The Validate procedure contains:

```

MAINTAIN FILE EMPLOYEE FROM EMPSTACK
INFER EMP_ID INTO EMPSTACK;
COMPUTE CNT/I4=1;
REPEAT EMPSTACK.FOCCOUNT;
    IF EMPSTACK(CNT).CURR_SAL GT 100000 THEN BEGIN
        COMPUTE FOCERROR=CNT;
        GOTO EXITREPEAT;
    ENDBEGIN
    ELSE COMPUTE CNT=CNT+1;
ENDREPEAT
END

```

The PutData procedure, residing on a remote Reporting Server, contains:

```

MAINTAIN FILE EMPLOYEE FROM EMPSTACK
INFER EMP_ID INTO EMPSTACK;
FOR ALL INCLUDE EMP_ID CURR_SAL FROM EMPSTACK;
END

```

Example: Calling Procedures to Populate Stacks

The following example shows all of the models and body types for the displayed country and car. The first calls GETCARS to populate the stack containing Country and Car. Maintain Data then calls GETMODEL to populate the other stack with the proper information. Each time a new Country and Car combination is introduced, Maintain Data calls GETMODEL to repopulate the stack.

```

MAINTAIN FILE CAR
INFER COUNTRY CAR INTO CARSTK;
INFER COUNTRY CAR MODEL BODYTYPE INTO DETSTK;
CALL GETCARS INTO CARSTK;
PERFORM GET_DETAIL;
Winform Show CARFORM;

CASE GET_DETAIL
CALL GETMODEL FROM CARSTK INTO DETSTK;
ENDCASE

CASE NEXTCAR
IF CARSTK.FOCINDEX LT CARSTK.FOCCOUNT
    THEN COMPUTE CARSTK.FOCINDEX= CARSTK.FOCINDEX +1;
    ELSE COMPUTE CARSTK.FOCINDEX = 1;
PERFORM GET_DETAIL;
ENDCASE

CASE PREVCAR
IF CARSTK.FOCINDEX GT 1
    THEN COMPUTE CARSTK.FOCINDEX= CARSTK.FOCINDEX -1;
    ELSE COMPUTE CARSTK.FOCINDEX = CARSTK.FOCCOUNT;
PERFORM GET_DETAIL;
ENDCASE

```

The procedure GETCARS loads all Country and Car combinations into CARSTK.

```

MAINTAIN FILE CAR INTO CARSTK
FOR ALL NEXT COUNTRY CAR INTO CARSTK;
END

```

The procedure GETMODEL loads all model and body type combinations into CARSTK for displayed Country and Car combinations.

```

MAINTAIN FILE CAR FROM CARSTK INTO DETSTK
INFER COUNTRY CAR INTO CARSTK;
STACK CLEAR DETSTK;
REPOSITION COUNTRY;
FOR ALL NEXT COUNTRY CAR MODEL BODYTYPE INTO DETSTK
    WHERE COUNTRY EQ CARSTK(CARSTK.FOCINDEX).COUNTRY
    AND CAR EQ CARSTK(CARSTK.FOCINDEX).CAR;
END

```

CASE

The CASE command allows you to define a Maintain Data function. Maintain Data functions are sometimes also called cases. The CASE keyword defines the beginning of the function, and the ENDCASE keyword defines its end.

You can pass values to a Maintain Data function using its parameters, and you can pass values from a Maintain Data function using its parameters and its return value.

You can call a Maintain Data function in one of the following ways:

- ❑ Issuing a PERFORM or GOTO command.
- ❑ Calling the function directly.

Once control has branched to the function, it proceeds to execute the commands within it. If control reached the end of the function (that is, the ENDCASE command), it returns or exits depending on how the function was called:

- ❑ **Branch and return.** If the function was called by a branch-and-return command (that is, by a PERFORM command or an event handler), or called directly, control returns to the point immediately following the PERFORM, event handler, or function reference.
- ❑ **Branch.** If the function was called by a simple branch command, for example, a GOTO command, and control reaches the end of the function, it means that you have not provided any logic to direct control elsewhere and so it exits the procedure. If this is not the result you want, simply call the function using PERFORM instead of GOTO, or else issue a command before ENDCASE to transfer control elsewhere.

A CASE command that is encountered in the sequential flow of a procedure is not executed.

You assign a unique name to each function using the CASE command.

Syntax: How to Use the CASE Command

The syntax for the CASE command is:

```
CASE functionname [TAKES p1/t1[, ..., pn/tn]] [RETURNS result/t] [;  
  [declarations]  
  [commands]  
  .  
  .  
  .  
ENDCASE
```

where:

functionname

Is the name you give to the function, and it can be up to 66 characters long. The name must begin with a letter, and can include any combination of letters, digits, and underscores (_).

TAKES p1/t1

Specifies that the function takes parameters. *p1/t1...pn/tn* defines the parameters of the function (*p*) and the data type of each parameter (*t*). When you call the function, you pass it variables or constants to substitute for these parameters. Parameters must be scalar. They cannot be stacks.

If the function is the Top function or a task, it cannot take parameters.

RETURNS result/t

Specifies that the function returns a value. *result* is the name of the variable being returned, and *t* is the data type of the variable. The return value must be scalar. It cannot be a stack.

If the function is the Top function or a task, it cannot return a value.

declarations

Is an optional DECLARE command to declare any variables that are local to the function. These declarations must precede all other commands in the function.

commands

Is one or more commands, except for CASE, DESCRIBE, END, MAINTAIN, and MODULE.

i

Terminates the parameter and return variable definitions of the CASE command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Reference: Usage Notes for CASE

- The first function in a procedure must be an explicit or implicit Top function.
- CASE commands cannot be nested.

Reference: Commands Related to CASE

- ❑ **PERFORM.** Transfers control to another function. When control reaches the end of the function, it returns to the command following PERFORM.
- ❑ **GOTO.** Transfers control to another function or to the end of the current function. Unlike the PERFORM command, it does not return the control of the command that called the function.

Calling a Function: Flow of Control

When a function is called, and control in the function is complete, control returns to the next command after the call.

When the Increase function in the following example is complete, processing resumes with the line after the PERFORM command, the TYPE command:

```
PERFORM Increase;
TYPE "Returned from Increase";
.
.
.
CASE Increase
COMPUTE Salary = Salary * 1.05;
.
.
.
ENDCASE
```

Passing Parameters to a Function

In general, the parameters of a Maintain Data function are both input and output parameters:

- ❑ When one function calls another, the calling function passes the current values of the parameters.
- ❑ When the called function terminates, it passes back the current values of the parameters.

If the called function changes the values of any of its parameters, when it returns control to the calling function, the parameter variables in the calling function are set to those new values. The parameters are global to the calling and called functions.

This method of passing parameters is known as a call by reference, because the calling function passes a reference to the parameter variable (specifically, its address), not a copy of its value.

Note: There is one exception to this behavior. If you declare a function parameter (in the Function Editor or a CASE command) with one data type, but at run time you pass the function a value of a different data type, the value of the parameter is converted to the new data type. Data types, in this context, refer to basic data types, such as fixed-length character (A_n where n is greater than zero (0)), variable-length character (A_0), text, date, date-time, integer, single-precision floating point, double-precision floating point, 8-byte packed decimal, and 16-byte packed decimal. Other data attributes, such as length, precision, MISSING, and display options, can differ without causing a conversion. Any changes that the called function makes to the value of the parameter will not get passed back to the calling function. The parameter is local to the called function.

This method of passing parameters is known as a call by value, because the calling function passes a copy of the value of the parameter variable, not a pointer to the actual parameter variable itself.

Note that you should not pass a constant as a function parameter if the function may change the value of that parameter.

Using the Return Value of a Function

If a function returns a value using the RETURNS phrase, you can call that function anywhere you can use an expression. For example:

```

MAINTAIN FILE HousePlan
.
.
.
COMPUTE ConferenceRoom/D6.2 = FindArea(CRlength,CRwidth);
.
.
.
CASE FindArea TAKES Length/D6.2, Width/D6.2 RETURNS Area/D6.2;
Area = Length * Width;
ENDCASE
.
.
.
END

```

Using the Top Function

When you run a Maintain Data procedure, the procedure begins by executing its Top function. Every Maintain Data procedure has a Top function. Top does not take or return parameters. You can choose to define the Top function, beginning it with a CASE command and ending it with an ENDCASE command, as all other Maintain Data functions are defined. This is the recommended method for defining Top, and is how Maintain Data generates Top when creating a new procedure.

For example:

```
CASE Top
.
.
.
ENDCASE
```

COMMIT

The COMMIT command processes a logical transaction. A logical transaction is a group of data source operations in an application that are treated as one. The COMMIT operation signals a successful end of a transaction and writes the INCLUDE, UPDATE, and DELETE operations to the data source. The data source is, or should be, in a consistent state and all of the updates made by that transaction are now made permanent.

Syntax: How to Use the COMMIT Command

The syntax of the COMMIT command is

```
COMMIT [ ; ]
```

where:

```
;
```

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Language Rules Reference](#) on page 11.

Reference: Usage Notes for COMMIT

- ❑ When you issue a transaction that writes to multiple types of data sources, each DBMS evaluates its part of the transaction independently. When a COMMIT command ends the transaction, the success of the COMMIT against each data source type is independent of the success of the COMMIT against the other data source types.

For example, if you run a procedure that accesses the App Studio data sources Employee and JobFile and the SQL Server data source Salary, the success or failure of the COMMIT for Salary is independent of the success of the COMMIT for Employee and JobFile. This is known as a broadcast commit.

- ❑ COMMIT is automatically issued when a procedure does not contain any COMMIT commands, and the application is exited normally. This means an error did not cause program termination. If a procedure does not contain any COMMIT commands and it is terminated abnormally (for example, if the system has run out of memory), a COMMIT is not issued. When a called procedure is exited, an automatic COMMIT is not issued. COMMIT is only issued when exiting the application.
- ❑ The variable FocCurrent is set after a COMMIT finishes. If the COMMIT is successful, FocCurrent is set to zero (0). If FocCurrent is non-zero, the COMMIT failed, and all of the records in the logical unit of work will be rolled back because an internal ROLLBACK will be issued.

COMPUTE

The COMPUTE command enables you to:

- ❑ Create a global variable, including global objects, and optionally assign it an initial value. You can use the DECLARE command to create both local and global variables. See [Local and Global Declarations](#) on page 80 for more information about local and global variables.
- ❑ Assign a value to an existing variable.
- ❑ Dynamically change the property of an object.

Syntax: How to Use the COMPUTE Command

The syntax of the COMPUTE command is:

```
[COMPUTE]
target_variable[/datatype [DFC cc YRT yy] [missing]][= expression];
.
.
.
missing: [MISSING {ON|OFF} [NEEDS] [SOME|ALL] [DATA]]
```

where:

COMPUTE

Is an optional keyword. It is required if the preceding command can take an optional semicolon (;) terminator, but was coded without one. In all other situations, it is unnecessary.

When the COMPUTE keyword is required, and there is a sequence of COMPUTE commands, the keyword needs to be specified only once for the sequence, for the first command in the sequence.

target_variable

Is the name of the variable which is being created and/or to which a value is being assigned. A variable name must start with a letter and can only contain letters, numbers, and underscores (_).

datatype

Is included in order to create a new variable. If you are creating a simple variable, you can specify all built-in formats and edit options, except for TX, as described for the Master File FORMAT attribute in the *Describing Data With WebFOCUS Language* manual. If you are creating an object, you can specify a class. You must specify a data type when you create a new variable. You can only specify the data type of a variable once, and you cannot redefine the data type of an existing variable.

DFC *cc*

Specifies a default century that will be used to interpret any dates with unspecified centuries in expressions assigned to this variable. *cc* is a two-digit number indicating the century (for example, 19 would indicate the twentieth century). If this is not specified, it defaults to 19.

Specifying DFC *cc* is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

YRT *yy*

Specifies a default threshold year for applying the default century identified in DFC *cc*. *yy* is a two-digit number indicating the year. If this is not specified, it defaults to 00.

When the year of the date being evaluated is less than the threshold year, the century of the date being evaluated defaults to the century defined in DFC *cc* plus one. When the year is equal to or greater than the threshold year, the century of the date being evaluated defaults to the century defined in DFC *cc*.

Specifying YRT *yy* is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

missing

Is used to allow or disallow null values. This is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

MISSING

If the MISSING syntax is omitted, the default value of the variable is zero (0) for numeric variables and a space for character and date and time variables. If it is included, its default value is null.

ON

Sets the default value to null.

OFF

Sets the default value to zero (0) or a space.

NEEDS

Is an optional keyword that clarifies the meaning of the command for a reader.

SOME

Indicates that for the target variable to have a value, some (at least one) of the variables in the expression must have a value. If all of the variables in the expression are null, the target variable will be null. This is the default.

ALL

Indicates that for the target variable to have a value, all the variables in the expression must have values. If any of the variables in the expression are null, the target variable will be null.

DATA

Is an optional keyword that clarifies the meaning of the command for a reader.

=

Is optional when COMPUTE is used solely to establish format. The equal sign is required when *expression* is used.

expression

Is any standard Maintain Data expression, as defined in [Expressions Reference](#) on page 21. Each expression must end with a semicolon (;). When creating a new variable using a class data type, you must omit *expression*.

Example: Moving the COMPUTE Keyword

You can place an expression on the same line as the COMPUTE keyword, or on a different line, so that

```
COMPUTE  
TempEmp_ID/A9 = '000000000';
```

is the same as:

```
COMPUTE TempEmp_ID/A9 = '000000000';
```

Example: Multi-Statement COMPUTE Commands

You can type a COMPUTE command over as many lines as you need. You can also specify a series of assignments as long as each expression is ended with a semicolon (;). For example:

```
COMPUTE TempEmp_ID/A9 = '000000000';
      TempLast_Name/A15 ;
      TempFirst_Name/A10;
```

Example: Combining Several Statements Into One Line

Several expressions can be placed on one line as long as each expression ends with a semicolon (;). The following shows two COMPUTE expressions on one line and a third COMPUTE on the next line. The first computes a five percent raise and the second increases education hours by eight. The third concatenates two name fields into one field:

```
COMPUTE Raise/D12.2=Curr_Sal*1.05; Ed_Hrs=Ed_Hrs+8;
Name/A25 = First_Name || Last_Name;
```

Reference: Usage Notes for COMPUTE

- ❑ If the names of incoming data fields are not listed in the Master File, they must be defined before they can be used. Otherwise, rejected fields are unidentified and the procedure terminates.

There are two different ways these fields can be defined. The first uses the notation:

```
COMPUTE target_variable/format =;
```

Because there is no expression after the equal sign (=), the field and its format is made known, but nothing else happens. If this style is used for a field in a form, the field appears on the form without a default value. Because COMPUTE is used solely to establish format, the equal sign is optional and the following syntax is also correct:

```
COMPUTE target_variable/format;
```

The second method of defining a user-defined field can be used when an initial value is desired. The syntax is:

```
COMPUTE target_variable/format = expression;
```

- ❑ Each field referred to or created in a Maintain Data procedure counts as one field toward the 3,072 field limit, regardless of how often its value is changed by COMPUTE commands. However, if a data source field is read by a Winform command and also has its value changed by a COMPUTE command, it counts as two fields.

Reference: Commands Related to COMPUTE

- ❑ **DEFINE.** Is a Master File attribute (not a command) that defines temporary fields and derives their values from other fields in the data source. This type of temporary field is called a virtual field. DEFINE automatically creates a corresponding virtual column in every stack that includes the segment of the field. For more information, see the *Describing Data With WebFOCUS Language* manual.
- ❑ **DECLARE.** Creates local and global variables.

Using COMPUTE to Call Functions

When you call a function as a separate statement (that is, outside of a larger expression), if the preceding command can take an optional semicolon (;) terminator, but was coded without one, you must call the function in a COMPUTE or PERFORM command. You can use PERFORM for Maintain Data functions only, though not for Maintain Data functions that return a value. For example, in the following source code, the NEXT command does not end with a semicolon (;), so the function that follows it must be called in a COMPUTE command:

```
NEXT CustID INTO CustStack  
COMPUTE VerifyCustID();
```

However, in all other situations, you can call functions directly, without a COMPUTE command. For example, in the following source code, the NEXT command ends with a semicolon (;), so the function that follows it can be called without a COMPUTE command:

```
NEXT CustID INTO CustStack;  
VerifyCustID();
```

For more information about terminating commands with a semicolon (;), see [Terminating Command Syntax](#) on page 17.

Using COMPUTE to Dynamically Change the Property of an Object

Instead of using WINFORM SET to dynamically change the property of an object, compute a variable to a value, and assign that variable to the property of the object. When the variable is evaluated, the property of the object is dynamically set.

For example, instead of issuing the following WINFORM SET command:

```
WINFORM SET FORM.OBJECT.FOCUS TO HERE;
```

Issue the following COMPUTE command and assign the variable to the FOCUS property of the object.

```
COMPUTE VAR/I1=1;
```


where:

VAR

Is the name of the variable used in the dynamic set.

Note: In App Studio Maintain Data, assign the variable to the FOCUS property of the desired object.

COPY

The COPY command copies some or all of the rows of one stack into another stack. You can use the COPY command to overwrite existing rows in the target stack, to add new rows, or to create the entire target stack.

You must define the contents of a stack before copying data into it. This can be accomplished by issuing a NEXT or an INFER command for data source fields, and COMPUTE for non-data source fields.

Source and target database stacks used in the COPY command must be derived from the same data source description. The COPY command checks that the data source and segment names are the same, and copies all columns in the source stack whose names and data types exactly match columns in the target stack. In this context, data type refers to the basic data type (such as integer) and all other data attributes including length, precision, null (MISSING), and display options such as zero (0) suppression. Source and target columns do not need to be in the same sequence.

Syntax: How to Use the COPY Command

The syntax of the COPY command is

```
[FOR {int|ALL}|STACK] COPY FROM {stk(row)|CURRENT}
INTO {stk(row)|CURRENT} [WHERE expression] [;]
```

where:

FOR

Is a prefix used with *int* or ALL to specify the number of rows to copy from the source (FROM) stack into the target (INTO) stack. If you omit both FOR and STACK, only the first row of the source stack is copied.

int

Is an integer expression that specifies how many source stack rows to copy into the target stack. If *int* exceeds the number of source stack rows between the starting row and the end of the stack, all of those rows are copied.

ALL

Indicates that all of the rows starting with either the first row or the subscripted row are copied from the source (FROM) stack into the target (INTO) stack.

STACK

Is a synonym for the prefix FOR ALL. If you omit both FOR and STACK, only the first row of the source stack is copied.

FROM

Is used with a stack name to specify the stack from which to copy the data.

INTO

Is used with a stack name to specify the stack to be created or modified.

stk

Is the name of the source or target stack. You can specify the same stack as the source and target stacks.

row

Is a stack subscript that specifies a starting row number. It can be a constant, an integer variable, or any Maintain Data expression that results in an integer value. If you omit *row*, it defaults to 1.

CURRENT

Specifies the Current Area. If you specify CURRENT for the source stack, all Current Area fields that also exist in the target stack are copied to the target stack. You cannot specify CURRENT if you specify FOR or STACK.

WHERE

Specifies selection criteria for copying stack rows. If you specify a WHERE phrase, you must also specify a FOR or STACK phrase.

expression

Is any Maintain Data expression that resolves to a Boolean expression. Unlike an expression in the WHERE phrase of the NEXT command, it does not need to refer to a data source field.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Copying All Rows of a Stack

The following example copies the entire Emp stack into a new stack called Newemp:

```
FOR ALL COPY FROM Emp INTO Newemp;
```

Example: Copying a Specified Number of Stack Rows

The following example copies 100 rows from the Emp stack starting with row number 101. The rows are inserted beginning with row one of the stack Subemp:

```
FOR 100 COPY FROM Emp(101) INTO Subemp;
```

Example: Copying the First Row of a Stack

The following example copies the first row of the Emp stack into the first row in the Temp stack. Only the first row in the source (FROM) stack is copied because this is the default when a prefix is not specified for the COPY command. The data is copied into the first row of the Temp stack because the first row is the default when a row number is not supplied for the target (INTO) stack:

```
COPY FROM Emp INTO Temp;
```

Example: Copying a Row Into the Current Area

The following example copies the tenth row of the Emp stack into the Current Area. Only one row is copied from the Emp stack because the COPY command does not have a prefix. Every column in the stack is copied into the Current Area. If there is already a field in the Current Area with the same name as a column in the stack, the Current Area variable is replaced with data from the Emp stack:

```
COPY FROM Emp(10) INTO CURRENT;
```

Example: Copying Rows Based on Selection Criteria

You can also copy selected rows based on selection criteria. The following example copies every row in the World stack that has a Country equal to USA into a new stack called USA:

```
FOR ALL COPY FROM World INTO USA WHERE Country EQ 'USA';
```

The following takes data from one stack and places it into three different stacks: one to add data, one to change data, and one to update data.

```

FOR ALL COPY FROM Inputstk INTO Addstk WHERE Flag EQ 'A';
FOR ALL COPY FROM Inputstk INTO Delstk WHERE Flag EQ 'D';
FOR ALL COPY FROM Inputstk INTO Chngstk WHERE Flag EQ 'C';
FOR ALL INCLUDE Dbfield FROM Addstk;
FOR ALL DELETE Dbfield FROM Delstk;
FOR ALL UPDATE Dbfield1 Dbfield2 FROM Chngstk;

```

Example: Appending One Stack to Another

The following example takes an entire stack and adds it to the end of an existing stack. The subscript consists of an expression. Yeardata.FocCount is a stack variable where Yeardata is the name of the stack and FocCount contains the number of rows currently in the stack. By adding one to FocCount, the data is added after the last row:

```
FOR ALL COPY FROM Junedata INTO Yeardata(Yeardata.FocCount+1);
```

Reference: Usage Notes for COPY

- If the FOR *int* prefix specifies more rows than are in the source (FROM) stack, all of the rows are copied.
- Only the first row of the source (FROM) stack is copied if the COPY command does not include FOR.
- The entire stack is copied if the source (FROM) stack is not subscripted and FOR ALL is used.
- The row to start copying from defaults to the first row unless the source (FROM) stack is subscripted. If the source (FROM) stack is subscripted, the copy process starts with the row number and copies as many rows as specified in the FOR *n* prefix, or the remainder of the stack if FOR ALL is specified.
- No change is made to the source (FROM) stack unless it is also the target (INTO) stack.
- INTO CURRENT cannot be used with the FOR phrase and generates an error if specified.
- To copy an entire stack, specify FOR ALL without a subscripted source (FROM) stack.
- Stack columns created using the COMPUTE command cannot be copied into the Current Area.
- If the source (FROM) stack is the Current Area, the only Current Area fields that are copied are those that have a corresponding column name in the target (INTO) stack.
- If the target (INTO) stack is not subscripted, the data is copied into the first row in the stack. If the target (INTO) stack is subscripted, the copied row or rows are inserted at this row.

- ❑ If the COPY command specifies the command output destination as a row or rows of an existing stack that already have data in them, then the old data in these rows is overwritten with the new data when the COPY is executed.
- ❑ If the source (FROM) stack has fewer columns than the target (INTO) stack, the columns that do not have any data are initialized to blank, zero (0) , or null (missing) as appropriate.
- ❑ Source (FROM) stack rows will overwrite the specified target (INTO) stack rows if they already exist.
- ❑ If the COPY command creates rows in the target (INTO) stack, and the target (INTO) stack contains columns that are not in the source (FROM) stack, those columns in the new rows will be initialized to their default values of blank, zero (0), or null (missing).
- ❑ If the source (FROM) stack has more columns than the target (INTO) stack, only corresponding columns are copied.
- ❑ The FOR prefix copies rows from the source (FROM) stack one row at a time, not all at the same time. For example, the following command:

```
FOR ALL COPY FROM Car(Car.FocIndex) INTO Car(Car.FocIndex+1);
```

copies the first row into the second, then copies those same values from the second row into the third, and keeps going. When the command has finished executing, all rows will have the same values as the first row.

Reference: Commands Related to COPY

- ❑ **INFER.** Defines the columns in a stack.
- ❑ **COMPUTE.** Defines the columns in a stack for non-data source fields.
- ❑ **NEXT.** Defines the columns in a stack and places data into it.

DECLARE

The DECLARE command creates global and local variables (including objects), and gives you the option of assigning an initial value.

Where you place a DECLARE command within a procedure depends on whether you want it to define local or global variables. See [Local and Global Declarations](#) on page 80 for more information.

Syntax: How to Use the DECLARE Command

The syntax of the DECLARE command is

```

DECLARE
[ (
  objectname/datatype [DFC cc YRT yy] [missing]] [= expression];
.
.
.
[ )]
  missing: [MISSING {ON|OFF} [NEEDS] [SOME|ALL] [DATA]]

```

where:

objectname

Is the name of the object or other variable that you are creating. The name is subject to the standard naming rules of the Maintain Data language. See [Specifying Names](#) on page 13 for more information.

datatype

Is a data type (a class or built-in format).

expression

Is an optional expression that will provide the initial value of the variable. If the expression is omitted, the initial value of the variable is the default for that data type: a space or null for character and date and time data types, and zero (0) or null for numeric data types. When declaring a new variable using a class data type, you must omit *expression*.

DFC *cc*

Specifies a default century that will be used to interpret any dates with unspecified centuries in expressions assigned to this variable. *cc* is a two-digit number indicating the century (for example, 19 would indicate the twentieth century). If this is not specified, it defaults to 19.

This is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

YRT *yy*

Specifies a default threshold year for applying the default century identified in DFC *cc*. *yy* is a two-digit number indicating the year. If this is not specified, it defaults to 00.

When the year of the date being evaluated is less than the threshold year, the century of the date being evaluated defaults to the century defined in DFC *cc* plus one. When the year is equal to or greater than the threshold year, the century of the date being evaluated defaults to the century defined in DFC *cc*.

This is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

missing

Is used to allow or disallow null values. This is optional if the data type is a built-in format. It is not specified if the data type is a class, as it is relevant only for scalar variables.

MISSING

If the MISSING syntax is omitted, the default value of the variable is zero (0) for numeric variables and a space for character and date and time variables. If it is included, its default value is null.

ON

Sets the default value to null.

OFF

Sets the default value to zero (0) or a space.

NEEDS

Is an optional keyword that clarifies the meaning of the command for a reader.

SOME

Indicates that for the target variable to have a value, some (at least one) of the variables in the expression must have a value. If all of the variables in the expression are null, the target variable will be null. This is the default.

ALL

Indicates that for the target variable to have a value, all the variables in the expression must have values. If any of the variables in the expression is null, the target variable will be null.

DATA

Is an optional keyword that clarifies the meaning of the command for a reader.

()

Groups a sequence of declarations into a single DECLARE command. The parentheses are required for groups of local declarations, otherwise they are optional.

Reference: Commands Related to DECLARE

- DESCRIBE.** Defines classes and data type synonyms.
- COMPUTE.** Creates global variables, including objects, and assigns values to existing variables.

Local and Global Declarations

When you declare a new variable, you choose between making the variable:

- ❑ **Local.** To declare a local variable, issue the DECLARE command inside the desired function. The DECLARE command must precede all other commands in the function. A local variable is known only to the function in which it is declared.

To declare a local variable in the Top function, note that you cannot issue a DECLARE command in an implied Top function, but you can issue it within an explicit Top function.

- ❑ **Global.** To declare a global variable, place the DECLARE command outside of a function (for example, at the beginning of the procedure prior to all functions), or define it using the COMPUTE command anywhere in the procedure. Note that if you place any DECLARE commands at the beginning of the procedure, you must have an explicit Top case in order to end the global declarations. A global variable is known to all the functions in the procedure.

We recommend declaring your variables locally, and to work with a variable outside the function in which it was declared, passing it to the other function as an argument. Local variables are preferable to global variables because they are protected from unintended changes made in other functions.

DELETE

The DELETE command identifies segment instances from a transaction source (a stack or the Current Area) and deletes the corresponding instances from the data source.

When you issue the command, you specify an anchor segment. For each row in the transaction source, DELETE searches the data source for a matching segment instance. When it finds a match, it deletes that anchor instance and all the descendants of the anchor.

If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor, or provide ancestor segment key values in the source stack. This ensures that DELETE can navigate from the root to the first instance of the anchor segment.

Syntax: How to Use the DELETE Command

The syntax of the DELETE command is

```
[FOR {int|ALL}] DELETE segment [FROM stack(row)] [;]
```


where:

FOR

Is used with ALL or an integer to specify how many stack rows to use to identify segment instances. If FOR is omitted, one stack row will be used.

When you specify FOR, you must also specify FROM to identify a source stack.

int

Is an integer constant or variable that indicates the number of stack rows to use to identify segment instances to be deleted.

ALL

Specifies that the entire stack is used to delete the corresponding records in the data source.

segment

Specifies the anchor segment of the path to delete. To specify a segment, provide the name of the segment or of a field within the segment.

FROM

Is used to specify a stack whose key columns identify records to delete. If no stack is specified, data from the Current Area is used.

stack

Is a stack name. Only one stack can be specified.

row

Is a subscript that specifies which stack row to begin with.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Specifying Which Segments to Delete

The DELETE command removes the lowest specified segment and all of its descendant segments. For example, if a data source structure has four segments in a single path (named First, Second, Third, and Fourth), the command

```
DELETE First.Field1 Second.Field2;
```

will delete instances from the Second, Third, and Fourth segments.

If you issue the command

```
DELETE First.Field1;
```

you will delete the entire data source path.

Example: Deleting Records Identified in a Stack

In the following example, the data in rows 2, 3, and 4 of the Stkemp stack is used to delete data from the data source. The stack subscript indicates start in the second row of the stack and the FOR 3 means DELETE data in the data source based on the data in the next three rows.

```
FOR 3 DELETE Emp_ID FROM Stkemp(2);
```

Example: Deleting a Record Identified in a Form

The first example prompts the user for the employee ID in the EmployeeIDForm form. If the employee is already in the data source, all records for that employee are deleted from the data source. This includes the employee instance in the root segment and all descendent instances, such as pay dates, addresses, and so on. In order to find out if the employee is in the data source, a MATCH command is issued:

```
MAINTAIN FILE Employee  
WinForm Show EmployeeIDForm;  
CASE DELEMP  
MATCH Emp_ID;  
ON MATCH DELETE Emp_ID;  
ON NOMATCH TYPE "Employee id <Emp_ID not found. Reenter";  
COMMIT;  
ENDCASE  
END
```

When the user presses Enter, function DELEMP is triggered as an event handler from a form. Control is then passed back to EmployeeIDForm.

The second example provides the same functionality. The only difference is that a MATCH is not used to determine whether the employee already exists in the data source. The DELETE can only work if the record exists. Therefore, if an employee ID is entered that does not exist, the only action that can be taken is to display a message. In this case, the variable FocError is checked. If FocError is not equal to zero (0), then the DELETE failed and a message displays:

```

MAINTAIN FILE Employee
INFER EMP_ID INTO STACKEMP
Winform Show EmployeeIDForm;
CASE DELEMP
DELETE Emp_ID;
IF FocError NE 0 THEN
  TYPE "Employee id <Stackemp.Emp_ID not found.  Reenter";
COMMIT;
ENDCASE
END

```

Reference: Usage Notes for DELETE

- ❑ Because the DELETE command removes the instance pointed to by the segment position marker, after the deletion, the marker has a null value and the segment has no current position. To reestablish position, you can issue the REPOSITION command.
- ❑ You delete a unique segment by deleting its parent. To erase the fields of a unique segment without affecting its parent, you can instead update its fields to space, zero (0), or null.
- ❑ In order for the DELETE to work, the data must exist in the data source. When a set of rows are changed without first finding out if they already exist in the data source, then it is possible that some of the rows in the stack will be rejected. Upon the first rejection, the process stops and the rest of the set is rejected. If you want all rows to be accepted or rejected as a unit, you should treat the stack as a logical transaction, evaluate the FocError transaction variable, and then issue a ROLLBACK command if the entire stack is not accepted. The transaction variable FocErrorRow is automatically set to the number of the first row that failed.
- ❑ After the DELETE is processed, the transaction variable FocError is given a value. If the DELETE is successful, FocError is zero (0). If the DELETE fails (for example, the key values do not exist in the data source), FocError is set to a non-zero value and (if the DELETE is set-based) FocErrorRow is set to the number of the row that failed. If there is a concurrency conflict at COMMIT time, the transaction variable FocCurrent is set to a non-zero value.
- ❑ A DELETE command cannot have more than one input (FROM) stack.
- ❑ When a DELETE command is complete, the variable FocError is set. If the DELETE is successful (the records to be deleted exist in the data source), then FocError is set to zero (0). If the records do not exist, FocError is set to a non-zero value. If the DELETE operation was set-based, Maintain Data sets FocErrorRow to the number of the row that failed.
- ❑ Maintain Data requires that the data sources to which it writes have unique keys.

Reference: Commands Related to DELETE

- ❑ **COMMIT.** Makes all data source changes since the last COMMIT permanent.
- ❑ **ROLLBACK.** Cancels all data source changes made since the last COMMIT.

DESCRIBE

The DESCRIBE command enables you to define and work with objects (called classes) that cannot be defined using the standard set of classifications. Standard classifications include data types such as Alphanumeric, Numeric, and Integer (a subset of Numeric).

A class can represent a data type synonym. You can assign a name to a specific data type and then use this name as the format specification for variables. In this way, you can change the formats of multiple variables by changing the class definition.

A class can also represent an object consisting of other objects, called components of the class. You can define two types of components for a class, fields and functions (cases).

The DESCRIBE command defines the structure and behavior of a class. You then use the COMPUTE or DECLARE command to create an instance of the class. The COMPUTE command defines a global instance of the class. To create an instance that is local to a specific case, use the DECLARE command within that case.

To reference a component of a class instance, qualify the name of the component with the class instance name, separating them with a period (.). For example, consider a class called *Room*, which has components *length* and *width*. You can create a class instance named *MyRoom* in a COMPUTE or DECLARE command. For example:

```
COMPUTE MyRoom/Room;
```

To reference the length component of the MyRoom instance, qualify the component name with the class instance name:

```
MyRoom.length
```

This is similar to qualifying a field name in a data source with its file or segment name.

Within the DESCRIBE and ENDDDESCRIBE commands that define a class, you do not qualify component names for that class.

Syntax: How to Use the DESCRIBE Command

You must issue the DESCRIBE command outside of a function (case), for example, at the beginning of the procedure prior to all functions.

The syntax of the DESCRIBE command to define a new class is

```
DESCRIBE classname = ([superclass +] memvar/type [, memvar/type] ...) [;]
[memfunction [memfunction] ...
ENDDESCRIBE]
```

where:

classname

Is the name of the class that you are defining. The name is subject to the standard naming rules of the Maintain Data language. See [Specifying Names](#) on page 13 for more information.

superclass

Is the name of the superclass from which to derive this class. Used only to describe a subclass.

memvar

Names one of the member variables of the class. The name is subject to the standard naming rules of the Maintain Data language. See [Specifying Names](#) on page 13 for more information.

type

Is a data type (a built-in format or a class).

memfunction

Defines one of the member functions of the class. Member functions are defined the same way as other Maintain Data functions, using the CASE command. See [CASE](#) on page 63 for more information.

i

For class definitions, this terminates the definition if the definition omits member functions. If it includes member functions, the semicolon is omitted and the ENDDESCRIBE command is required.

For synonym definitions, this terminates the definition and is required.

ENDDESCRIBE

Ends the class definition if it includes member functions. If it omits member functions, the ENDDESCRIBE command must also be omitted, and the definition terminates with a semicolon (;).

The syntax of the DESCRIBE command to define a synonym for a data type is

```
DESCRIBE synonym = datatype ;
```

where:

synonym

Is a synonym for a data type (a class or format). The synonym is subject to the standard naming rules of the Maintain Data language. See [Specifying Names](#) on page 13 for more information.

;

For class definitions, this terminates the definition if the definition omits member functions. If it includes member functions, the semicolon is omitted and the ENDDESCRIBE command is required.

For synonym definitions, this terminates the definition and is required.

Example: Data Type Synonyms

Data type synonyms can make it easier for you to maintain variable declarations. For example, if your procedure creates many variables for names of people, and defines them all as A30, you would define a data type synonym for A30:

```
DESCRIBE NameType = A30;
```

You would then define all of the name variables as NameType:

```
DECLARE UserName/NameType;
COMPUTE ManagerName/NameType;
DECLARE CustomerName/NameType;
```

To change all name variables to A40, you could change all of them at once simply by changing one data type synonym:

```
DESCRIBE NameType = A40;
```

Example: Defining a Class and Creating an Instance

The following DESCRIBE command defines a class named Room in an architecture application. The components of the class are three fields, Width, Height, and Length:

```
DESCRIBE Room = (Width/I4, Height/I4, Length/I4);
```

The following COMPUTE command creates an instance of the Room class named *abc* and assigns values to the components, qualifying each component with the class name:

```
COMPUTE abc/Room;
abc.Width = 10;
abc.Height = 20;
abc.Length = 30;
```

Once the instance is created, you can use it in other Maintain Data commands. For example, the following TYPE command types each component value:

```
TYPE "Width=<abc.Width Height=<abc.Height Length=<abc.Length"
```

Class Member Functions

Functions included within a class definition specify operations that can be performed using the components of the class.

Two function names, `Startup` and `CleanUp`, are reserved and have specific uses.

If you define a case called `Startup`, that case is executed whenever an instance of the class is created. A global instance is created at the beginning of the Maintain Data procedure. A local instance is created each time the case in which it is declared is performed.

If you define a case called `CleanUp`, that case is executed whenever an instance of the class reaches the end of its scope. The scope of a global instance ends after execution of the Maintain Data procedure. The scope of a local instance ends each time execution returns to the procedure that performed the case in which it was declared.

Reference: Startup Case Considerations

You can create a global instance of a class using the `COMPUTE` command anywhere in the Maintain Data procedure. To create an instance local to a specific case, use the `DECLARE` command within that case.

You can use the `Startup` case to assign initial values to the components of a global instance of a class.

To pass initial values for class components to the `Startup` case:

- ❑ Define the `Startup` case to take arguments representing those components (with argument names different from the component names).
- ❑ In the `Startup` case, assign the incoming parameter values to the component field names.
- ❑ Then, in the `COMPUTE` command that creates the instance, specify argument values to pass to case `Startup`. For example, if the class is named `Room`, the instance is named `MyRoom`, and you want to assign component values `length=15` and `width= 10`, use the following syntax to pass the values to the `Startup` case:

```
COMPUTE MyRoom/Room ( 15, 10 ) ;
```

Note: The DECLARE command does not support passing arguments to the Startup case. However, you can always use a COMPUTE or DECLARE command to assign initial or non-initial values:

```
[COMPUTE|DECLARE] MyRoom/Room;  
MyRoom.length = 20;  
MyRoom.width = 15;
```

Reference: Executing Member Functions

Just as you can reference components of the class by qualifying the component names with the class instance name, you can execute a function within the class by qualifying the function name with the class instance name. For example, if the *Room* class contains a function called FINDAREA that takes the arguments *length* and *width* and returns the area, you can execute this function and type the returned value with the following commands:

```
AREA/I4 = MyRoom.FINDAREA(MyRoom.length, MyRoom.width)  
TYPE "AREA = <AREA>";
```

If the function operates on the components of the class, those components are available to the function without passing them as arguments. Therefore, if *length* and *width* are components of the *Room* class, the FINDAREA function does not need to take any arguments. In this case, you invoke the function as follows:

```
AREA/I4 = MyRoom.FINDAREA()  
TYPE "AREA = <AREA>";
```

Note that parentheses are required when invoking a member function even if the function does not take arguments.

Example: Defining a Class

The DESCRIBE command in the following Maintain Data procedure defines a class named Floor in an architecture application. The components of the class are three fields (Length, Width, and Area) and one case (PrintFloor). The COMPUTE command creates an instance of the class named MYFLOOR, assigns values to the components, and calls the PrintFloor function. Although the PrintFloor function does not take arguments, the parentheses are needed to identify PrintFloor as a function:


```

MAINTAIN
DESCRIBE Floor = (Length/I4, Width/I4, Area/I4)
  CASE PrintFloor
    TYPE "length=<Length width=<Width area=<Area";
  ENDCASE
ENDDESCRIBE

COMPUTE MYFLOOR/FLOOR;
MYFLOOR.Length = 15;
MYFLOOR.Width = 10;
MYFLOOR.Area = MYFLOOR.Length * MYFLOOR.Width;
MYFLOOR.PrintFloor();
END

```

The output is:

```
length=15 width=10 area=150
```

Example: Defining a Class With a Startup Case

The DESCRIBE command in the following Maintain Data procedure defines a class named Floor in an architecture application. The components of the class are three fields (Length, Width, and Area) and one case (PrintFloor). The COMPUTE command creates an instance of the class named MYFLOOR, passes values for the components to the Startup case, and calls the PrintFloor function. The Startup case initializes the component fields with the values passed in the COMPUTE command:

```

MAINTAIN
DESCRIBE Floor = (Length/I4, Width/I4, Area/I4)
  CASE Startup TAKES L/I4, W/I4, A/I4
    Length = L;
    Width = W;
    Area = A;
  ENDCASE
  CASE PrintFloor
    TYPE "In PrintFloor: length=<Length width=<Width area=<Area";
  ENDCASE
ENDDESCRIBE

COMPUTE MYFLOOR/FLOOR(15, 10, 150);
TYPE "After Startup: LENGTH=<MYFLOOR.LENGTH" |
  " WIDTH=<MYFLOOR.WIDTH AREA=<MYFLOOR.AREA";
MYFLOOR.PrintFloor();
END

```

The output is:

```
After Startup: LENGTH=15 WIDTH=10 AREA=150
In PrintFloor: length=15 width=10 area=150
```

Example: Defining and Using a Local Class Instance

In the following Maintain Data procedure, the DESCRIBE command defines a class named FNAME with components LAST and FIRST. The FORMNAME case concatenates the last and first names and separates them with a comma (,) and a space.

The main procedure loops through the first five records of the EMPLOYEE data source, and passes each last name and first name to case PRTFULL.

Case PRTFULL creates a local instance of the FNAME class, invokes the FORMNAME member function, and types the full name returned from that class. Although FORMNAME does not take any arguments, the parentheses used when invoking FORMNAME identify it as a function.

TYPE commands in each case illustrate the flow of control:

```

MAINTAIN FILE VIDEOTRK
  DESCRIBE FNAME = (LAST/A15, FIRST/A10)

  CASE STARTUP;
    TYPE "IN CASE STARTUP: I = <I";
  ENDCASE

  CASE FORMNAME RETURNS FULLNAME/A30;
    FULLNAME/A30 = LAST || ', ' | FIRST;
    TYPE "IN CASE FORMNAME: I = <I  FULLNAME = <FULLNAME";
  ENDCASE

  CASE CLEANUP;
    TYPE "IN CASE CLEANUP: I = <I";
  ENDCASE
ENDDESCRIBE

-* MAIN PROCEDURE

FOR 5 NEXT CUSTID INTO CUSTSTK;
REPEAT 5 I/I1 = 1;
COMPUTE LAST/A15 = CUSTSTK(I).LASTNAME;
COMPUTE FIRST/A10 = CUSTSTK(I).FIRSTNAME;
TYPE "IN MAIN PROCEDURE: I = <I  LAST = <LAST  FIRST = <FIRST";
PERFORM PRTFULL(LAST, FIRST);
ENDREPEAT I = I+1;

  CASE PRTFULL TAKES LAST/A15, FIRST/A10;
  -* MEMNAME IS A LOCAL VARIABLE
    DECLARE MEMNAME/FNAME;
    MEMNAME.LAST=LAST;
    MEMNAME.FIRST=FIRST;
    NEWNAME/A30 = MEMNAME.FORMNAME();
    TYPE "IN CASE PRTFULL: I = <I  MEMBER NAME IS <NEWNAME";
  ENDCASE
END

```

The output shows that the Startup case is called prior to each invocation of case PRTFULL (which defines the local instance), and the Cleanup case is called at the end of each invocation of case PRTFULL:

```

IN MAIN PROCEDURE: I = 1 LAST = CRUZ FIRST = IVY
IN CASE STARTUP: I = 1
IN CASE FORMNAME: I = 1 FULLNAME = CRUZ, IVY
IN CASE PRTFULL: I = 1 MEMBER NAME IS CRUZ, IVY
IN CASE CLEANUP: I = 1
IN MAIN PROCEDURE: I = 2 LAST = HANDLER FIRST = EVAN
IN CASE STARTUP: I = 2
IN CASE FORMNAME: I = 2 FULLNAME = HANDLER, EVAN
IN CASE PRTFULL: I = 2 MEMBER NAME IS HANDLER, EVAN
IN CASE CLEANUP: I = 2
IN MAIN PROCEDURE: I = 3 LAST = WILSON FIRST = KELLY
IN CASE STARTUP: I = 3
IN CASE FORMNAME: I = 3 FULLNAME = WILSON, KELLY
IN CASE PRTFULL: I = 3 MEMBER NAME IS WILSON, KELLY
IN CASE CLEANUP: I = 3
IN MAIN PROCEDURE: I = 4 LAST = KRAMER FIRST = CHERYL
IN CASE STARTUP: I = 4
IN CASE FORMNAME: I = 4 FULLNAME = KRAMER, CHERYL
IN CASE PRTFULL: I = 4 MEMBER NAME IS KRAMER, CHERYL
IN CASE CLEANUP: I = 4
IN MAIN PROCEDURE: I = 5 LAST = GOODMAN FIRST = JOHN
IN CASE STARTUP: I = 5
IN CASE FORMNAME: I = 5 FULLNAME = GOODMAN, JOHN
IN CASE PRTFULL: I = 5 MEMBER NAME IS GOODMAN, JOHN
IN CASE CLEANUP: I = 5

TRANSACTIONS: COMMITS      =          1 ROLLBACKS =          0
SEGMENTS   : INCLUDED    =          0 UPDATED   =          0 DELETED   =          0

```

Example: Defining and Using a Global Class Instance

In the following Maintain Data procedure, the DESCRIBE command defines a class named FNAME with components LAST and FIRST. The FORMNAME case concatenates the last and first names and separates them with a comma (,) and a space.

The main procedure loops through the first five records of the EMPLOYEE data source, and passes each last name and first name to case PRTFULL.

Case PRTFULL creates a global instance of the FNAME class, invokes the FORMNAME member function, and types the full name returned from that class. Although FORMNAME does not take any arguments, the parentheses used when invoking FORMNAME identify it as a function.

TYPE commands in each case illustrate the flow of control:

DESCRIBE

```
MAINTAIN FILE VIDEOTRK
DESCRIBE FNAME = (LAST/A15, FIRST/A10)
CASE STARTUP TAKES LASTNAME/A15, FIRSTNAME/A10;
  TYPE "IN CASE STARTUP: I = <I  LAST = <LASTNAME  FIRST = <FIRSTNAME";
  LAST = LASTNAME;
  FIRST = FIRSTNAME;
ENDCASE

CASE FORMNAME RETURNS FULLNAME/A30;
  FULLNAME/A30 = LAST || ' ' | FIRST;
  TYPE "IN CASE FORMNAME: I = <I  FULLNAME = <FULLNAME";
ENDCASE

CASE CLEANUP;
  TYPE "IN CASE CLEANUP: I = <I  ";
ENDCASE
ENDDESCRIBE

-*MAIN PROCEDURE

FOR 5 NEXT CUSTID INTO CUSTSTK;

REPEAT 5 I/I1 = 1;
  COMPUTE LAST/A15 = CUSTSTK(I).LASTNAME;
  COMPUTE FIRST/A10 = CUSTSTK(I).FIRSTNAME;
  TYPE "IN MAIN PROCEDURE: I = <I  LAST = <LAST  FIRST = <FIRST";
  PERFORM PRTFULL(LAST, FIRST);
ENDREPEAT I = I+1;

CASE PRTFULL TAKES LAST/A15, FIRST/A10;
  COMPUTE MEMNAME/FNAME('ABEL', 'AARON');
  NEWNAME/A30 = MEMNAME.FORMNAME();
  TYPE "IN CASE PRTFULL: I = <I  MEMBER NAME IS <NEWNAME";
ENDCASE
END
```

The output shows that the Startup case is called at the start of the Maintain Data procedure, and the Cleanup case is called following the execution of the entire Maintain Data procedure:

```

IN CASE STARTUP: I = 0 LAST = ABEL FIRST = AARON

IN MAIN PROCEDURE: I = 1 LAST = CRUZ FIRST = IVY
IN CASE FORMNAME: I = 1 FULLNAME = CRUZ, IVY
IN CASE PRTFULL: I = 1 MEMBER NAME IS CRUZ, IVY
IN MAIN PROCEDURE: I = 2 LAST = HANDLER FIRST = EVAN
IN CASE FORMNAME: I = 2 FULLNAME = HANDLER, EVAN
IN CASE PRTFULL: I = 2 MEMBER NAME IS HANDLER, EVAN
IN MAIN PROCEDURE: I = 3 LAST = WILSON FIRST = KELLY
IN CASE FORMNAME: I = 3 FULLNAME = WILSON, KELLY
IN CASE PRTFULL: I = 3 MEMBER NAME IS WILSON, KELLY
IN MAIN PROCEDURE: I = 4 LAST = KRAMER FIRST = CHERYL
IN CASE FORMNAME: I = 4 FULLNAME = KRAMER, CHERYL
IN CASE PRTFULL: I = 4 MEMBER NAME IS KRAMER, CHERYL
IN MAIN PROCEDURE: I = 5 LAST = GOODMAN FIRST = JOHN
IN CASE FORMNAME: I = 5 FULLNAME = GOODMAN, JOHN
IN CASE PRTFULL: I = 5 MEMBER NAME IS GOODMAN, JOHN

TRANSACTIONS: COMMITS      =          1 ROLLBACKS =          0
SEGMENTS   : INCLUDED    =          0 UPDATED   =          0 DELETED   =          0

IN CASE CLEANUP: I = 6

```

Defining and Using Superclasses and Subclasses

After you describe a class, you can derive other classes from it. Subclasses inherit member variables and functions from their superclasses.

Order of classes matters when defining superclasses and subclasses. You must describe a superclass prior to its subclasses.

A class can also use another class as one of its components. Again, order matters. You must describe the class you are using as a component prior to the class that uses it.

Example: Defining a Subclass

The following example describes two classes, Floor and Room. Floor consists of components Length and Width and member function FloorArea.

Room is a subclass of Floor. It inherits components Length and Width and member function FloorArea. It adds component Depth and function RoomVolume.

The main procedure creates an instance of Room called MYROOM. It then assigns values to the components, including inherited components Length and Width. It invokes the inherited member function FloorArea as well as the RoomVolume function.

The main procedure then types the component values and the values returned by the member functions.

```
MAINTAIN

DESCRIBE Floor = (Length/I4, Width/I4)
CASE FloorArea RETURNS Area/I4;
  Area = Length * Width;
ENDCASE
ENDDESCRIBE

DESCRIBE Room = (Floor + Depth/I4)
CASE RoomVolume RETURNS Volume/I4;
  Volume = FLOORAREA() * Depth;
ENDCASE
ENDDESCRIBE

COMPUTE MYROOM/ROOM;
MYROOM.LENGTH = 15;
MYROOM.WIDTH = 10;
MYROOM.DEPTH = 10;
AREA/I4 = MYROOM.FLOORAREA();
VOLUME/I4 = MYROOM.RoomVolume();
TYPE "LENGTH=<MYROOM.Length, WIDTH=<MYROOM.Width, " |
      "DEPTH=<MYROOM.DEPTH, AREA=<AREA, " |
      "VOLUME=<VOLUME";
END
```

The output is:

```
LENGTH=15, WIDTH=10, DEPTH=10, AREA=150, VOLUME=1500
```

Example: Using a Class as a Component of Another Class

The following example describes three classes: RoomDetail, Floor, and Room.

RoomDetail has no member functions. It consists of two components, Depth and RmType.

Floor consists of components Length and Width and member function FloorArea.

Room is a subclass of Floor. It inherits components Length and Width and member function FloorArea. In addition, it has member function RoomVolume and component RmType, which is an instance of class RoomDetail. When referring to the components of RoomDetail, you must qualify them with their instance name. For example, the Rtype component is referenced as follows:

```
RmType.Rtype
```

The main procedure creates an instance of Room called MYROOM. It then assigns values to the components, including inherited components Length and Width. When assigning values to the components of the RmType instance of the RoomDetail class, it must qualify them with the instance name, MYROOM. Since these names already had one level of qualification when they were referenced in the Room class, they now have two levels of qualification. For example, the following assigns the value 10 to the Depth component:

```
MYROOM.RmType.Depth = 10;
```

Note that when no ambiguity in the variable name will result, only one level of qualification is actually enforced. MYROOM.Depth is understood as MYROOM.RmType.Depth since Depth does not appear in any other context.

The main procedure invokes the inherited member function FloorArea, as well as the RoomVolume function, then types the component values and the values returned by the member functions.

```
MAINTAIN
  DESCRIBE RoomDetail = (Depth/I4, Rtype/A10);

  DESCRIBE Floor = (Length/I4, Width/I4)
  CASE FloorArea RETURNS Area/I4;
    Area = Length * Width;
  ENDCASE
ENDDESCRIBE

  DESCRIBE Room = (Floor + RmType/RoomDetail)
  CASE RoomVolume RETURNS Volume/I4;
    Volume = FLOORAREA() * RmType.Depth;
    TYPE "ROOM TYPE IS <RmType.Rtype>";
  ENDCASE
ENDDESCRIBE

COMPUTE MYROOM/ROOM;
MYROOM.LENGTH = 15;
MYROOM.WIDTH = 10;
MYROOM.RmType.Depth = 10;
MYROOM.RmType.Rtype = 'DINING';
AREA/I4 = MYROOM.FLOORAREA();
VOLUME/I4 = MYROOM.RoomVolume();
TYPE "LENGTH=<MYROOM.LENGTH, WIDTH=<MYROOM.WIDTH, " |
"DEPTH=<MYROOM.rmtime.DEPTH, AREA=<AREA, " |
" VOLUME=<VOLUME";
END
```

The output is:

```
ROOM TYPE IS DINING
LENGTH=15, WIDTH=10, DEPTH=10, AREA=150, VOLUME=1500
```

END

Reference: Commands Related to DESCRIBE

- ❑ **DECLARE.** Creates local and global variables, including objects.
- ❑ **COMPUTE.** Creates global variables, including global objects, and assigns values to existing variables.

END

The END command marks the end of a Maintain Data procedure and terminates its execution.

Syntax: How to Use the END Command

The syntax of the END command is

```
END
```

where:

```
END
```

Is the last line of the procedure, and must be coded in uppercase letters.

Reference: Commands Related to END

- ❑ **MAINTAIN.** Is used to initiate the parsing and execution of a Maintain Data procedure.
- ❑ **CALL.** Is used to call one procedure from another.

EXEC

The EXEC command enables you to call an App Studio procedure and pass parameters to and from the procedure. You can run any App Studio procedure residing on a Reporting Server accessible to the Reporting Server where the calling procedure resides. From an App Studio procedure you can run many other types of procedures, including compiled C programs, CICS transactions, and native RDBMS command files.

Syntax: How to Use the EXEC Command

The syntax of the EXEC command is

```
EXEC progrname [KEEP|DROP] [PATH {VAR|LIST}] [FROM var_list] [INTO stacks]  
[;]
```

where:

```
progrname
```

Is the name of the external procedure residing on the remote Reporting Server.

KEEP | DROP

The DROP parameter terminates the server session. The KEEP parameter leaves the server session active for reuse by subsequent external procedures. KEEP is the default.

PATH

Is used to specify additional locations (search paths) the system should use when searching for dependent resources (Master Files, imported modules, and others). The path location names are application names existing within the APPROOT directory structure or application names that have been introduced with the APP MAP command. The search path value can be in the form of a Maintain Data variable or a list of literal values enclosed in double quotation marks ("), as follows:

```
EXEC Procedure PATH "AppDir1 AppDir2 AppDir3" ;
EXEC Procedure PATH MyVariable ;
```

FROM

Is included to pass one or more variables to the external procedure.

INTO

Is included to identify the data stack to receive the answer set or sets coming from the external procedure.

var_list

Is one or more scalar variables that you pass to the target procedure, where they are received as numbered amper variables. You can pass any scalar variable except for those defined as variable-length character variables (that is, except for those defined as AO or TX). Unlike the CALL command, you cannot pass stacks to the target procedure.

The length of a single parameter cannot exceed 32,000 characters. The total length of all specified parameters cannot exceed 32,000 characters.

stacks

Is one or more stacks, each of which will receive an answer set from the target procedure. To retrieve multiple answer sets, specify multiple stacks. The stacks are populated sequentially as each answer set is returned by the external procedure. You can pass any stack except for those defined using STACK OF.

The number of variables specified in the EXEC command must not exceed the number returned by the external procedure. If the number specified is fewer than the number returned, the extra returned parameters are ignored.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the semicolon, see [Terminating Command Syntax](#) on page 17.

FocCount

The FocCount stack variable contains the number of rows in the stack. In an empty stack, FocCount is 0. This variable is automatically maintained and the user does not need to do anything when new rows are added or deleted from the stack. For example, the following stack variable contains the number of rows in the EmplInfo stack:

```
EmplInfo.FocCount
```

The FocCount variable is useful as a test to see whether a data source retrieval command is successful. For example, after putting data into a stack, FocCount can be checked to see if its value is greater than zero (0). FocCount can also be used to perform an action on every row in a stack. A repeat loop can be set up to loop the number of times specified by the FocCount variable.

The following example computes a new salary for each row retrieved from the data source:

```
FOR ALL NEXT Emp_ID Curr_Sal INTO Pay;  
COMPUTE Pay.NewSal/D12.2=;  
REPEAT Pay.FocCount Cnt/I4=1;  
    COMPUTE Pay(Cnt).NewSal/D12.2 = Pay(Cnt).Curr_Sal * 1.05;  
ENDREPEAT Cnt=Cnt+1;
```

FocCurrent

FocCurrent contains the return code from logical transaction processing. This variable indicates whether or not there is a conflict with another transaction. If the variable value is zero (0), there is no conflict and the transaction is accepted. If the value is non-zero, there is a conflict. FocCurrent is set after each COMMIT and ROLLBACK command.

FocCurrent is local to a procedure. If you need a given FocCurrent value to be available to another procedure, you must pass it to that procedure as an argument.

FocError

FocError contains the return code from the INCLUDE, UPDATE, and DELETE commands. If all the rows in the stack are successfully processed, FocError is set to zero (0). FocError is set to a non-zero value if:

- ❑ INCLUDE rejects the input.

- ❑ UPDATE rejects the update.
- ❑ DELETE rejects the delete.
- ❑ REVISE rejects the changes.

FocError is a global variable. You do not need to pass it between procedures. Its value is cleared each time a Maintain Data procedure is called.

FocErrorRow

After any set-based data source operation (FOR ... UPDATE, DELETE, REVISE, or INCLUDE), if FocError is set to a non-zero value, then FocErrorRow is the number of the row that caused the error.

FocErrorRow is local to a procedure. If you need a given FocErrorRow value to be available to another procedure, you must pass it to that procedure as an argument.

FocFetch

FocFetch contains the return code of the most recently issued NEXT or MATCH command. If the NEXT or MATCH command returned data, FocFetch is set to zero (0). Otherwise, it is set to a non-zero value.

It is recommended that you test FocFetch in place of issuing the ON NEXT, ON NONEXT, ON MATCH, and ON NOMATCH commands. FocFetch accomplishes the same thing more efficiently.

For example:

```
FOR ALL NEXT CustID INTO CustOrderStack;
IF FocFetch NE 0 THEN ReadFailed();
```

FocFetch is local to a procedure. If you need a given FocFetch value to be available to another procedure, you must pass it to that procedure as an argument.

FocIndex

The FocIndex stack variable is a pointer to the current instance in a stack. In an empty stack, FocIndex is 1.

This variable is manipulated by the developer and can be used to do things such as determine which row of a stack is to be displayed on a form. A form displays data from a stack based on the value of FocIndex. For example, if a form currently displays data from the PayInfo stack and the following compute is issued:

```
COMPUTE PayInfo.FocIndex=15;
```

The fifteenth row of the stack displays in the form.

FocMsg

FocMsg is a system stack with one A80 column named Msg. When a Maintain Data procedure executes either an external procedure or a Maintain Data procedure on a remote server (that is, a Maintain Data procedure called using the CALL *procname* command), all of the messages that the called procedure writes to the default output device are automatically copied to the FocMsg stack of the calling procedure. This includes messages issued by TYPE and SAY commands that do not specify a file, and informational and error messages.

If the external procedure calls other external procedures, all messages posted by the chain of external procedures are copied to the same FocMsg stack in the calling Maintain Data procedure. Non-App Studio logic (such as a compiled 3GL program or a CICS transaction) that is called from an external procedure does not copy to FocMsg.

FocMsg is global to each Maintain Data procedure.

Example: **Cycling Through All the Messages in FocMsg**

You can use FocCount to cycle through all of the messages that have been posted to FocMsg:

```
COMPUTE Counter/I3=1;  
REPEAT FocMsg.FocCount;  
    TYPE "<FocMsg(Counter).Msg";  
    COMPUTE Counter=Counter+1;  
ENDREPEAT
```

Example: **Retrieving Messages Posted by an External Procedure**

This example illustrates how to retrieve messages that were posted by an external procedure.

Client Procedure

```

1.  MAINTAIN FILE MOVIES
2.  INFER MovieCode Title INTO MoviesInfo;
3.  EXEC GetMovie INTO MoviesInfo;
4.  COMPUTE I/I4=1;
5.  REPEAT 3;
6.  TYPE
7.  "Movie code is: << MoviesInfo(I).MovieCode"
8.  "      Title: << MoviesInfo(I).Title";
9.  COMPUTE I=I+1;
10. ENDREPEAT
11.
12. COMPUTE I=1;
13. REPEAT FocMsg.FocCount;
14. TYPE "Here are the messages from the server: <<FocMsg(I).Msg";
15. COMPUTE I=I+1;
16. ENDREPEAT
17. END

```

External procedure GetMovie

```

1.  TABLE FILE MOVIES
2.  PRINT MOVIECODE TITLE
3.  ON TABLE PCHOLD
4.  END
5.  RUN
6.  -TYPE "Finished with the movies retrieval"

```

GOTO

The GOTO command is used to transfer control to a different Maintain Data function, to a special point within the current function, or to terminate the application.

To transfer control to a different function, it is recommended that you use the PERFORM command instead of GOTO.

Syntax: How to Use the GOTO Command

The syntax of the GOTO command is

```
GOTO destination [;]
```

where:

destination

Is one of the following:

functionname

Specifies the name of the function to which control is transferred. Maintain Data expects to find a function by that name in the procedure. You cannot use GOTO with a function that has parameters.

Top

Transfers control to the beginning of the *Top* function. All local variables are freed and current data source positions are retained, as are any uncommitted data source transactions.

END [KEEP|RESET]

Terminates the procedure. Control returns to whatever called the procedure. No function may be named END, as such a function would be ignored and never executed.

KEEP

Terminates a called procedure, but keeps its data (the values of its variables and data source position pointers) in memory. It remains in memory through the next call or, if it is not called again, until the application terminates.

RESET

Terminates a called procedure and clears its data from memory. This is the default.

EXIT

This is similar to GOTO END but immediately terminates all procedures in an application. This means that if one procedure calls another and the called procedure issues a GOTO EXIT, both procedures are ended by the GOTO EXIT command. No function may be named EXIT.

ENDCASE

Transfers control to the ENDCASE command in the function, and the function is exited. For information about the ENDCASE command, see [CASE](#) on page 63.

ENDREPEAT

Transfers control to the ENDREPEAT command in the current REPEAT loop. The loop is not exited. All appropriate loop counters specified on the ENDREPEAT command are incremented. For information about the REPEAT and ENDREPEAT commands, see [REPEAT](#) on page 139.

EXITREPEAT

Exits the current REPEAT loop. Control transfers to the next line after the ENDREPEAT command. For information about the REPEAT and ENDREPEAT commands, see [REPEAT](#) on page 139.

;

Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

For example, to branch to the function named MainMenu, you would issue the command:

```
GOTO MainMenu
```

Reference: Usage Notes for GOTO

- If the GOTO specifies a function name that does not exist in the program, an error occurs at parse time, which occurs before execution.
- When one procedure calls another, and the called procedure has a GOTO END command, GOTO END ends only the called procedure. The calling procedure is unaffected. A GOTO END does not cause a COMMIT. This allows a called procedure to exit and have the calling program issue the COMMIT when appropriate. For information about the COMMIT command, see [COMMIT](#) on page 67.

Reference: Commands Related to GOTO

- PERFORM.** Control to another function. When the function finishes, control is returned to the command following the PERFORM.
- CASE/ENDCASE.** Allows a set of commands to be grouped together.
- REPEAT/ENDREPEAT.** Provides a general looping facility.

Using GOTO With Data Source Commands

A GOTO command can be executed in a MATCH command following an ON MATCH or ON NOMATCH command, or in NEXT following ON NEXT or ON NONEXT. The following syntax branches to the function MatchEdit when a MATCH occurs:

```
ON MATCH GOTO MatchEdit;
```

GOTO and ENDCASE

When control is transferred to a function with the GOTO command, every condition for exiting that function must contain a command indicating where control should be passed to. If an ENDCASE command is reached by either GOTO or normal program flow, and Maintain Data has not received any instructions as to where to go next, Maintain Data takes a default action and exits the procedure. ENDCASE is treated differently when GOTO and PERFORM are combined. See [PERFORM](#) on page 137 for more information.

GOTO and PERFORM

It is recommended that you do not issue a GOTO command within the scope of a PERFORM command.

The scope of a PERFORM command extends from the moment at which it is issued to the moment at which control returns normally to the command or form control point immediately following it. The scope includes any additional PERFORM commands nested within it.

For example, if the Top function issues a PERFORM command to call Case One, Case One issues a PERFORM command to call Case Two. Case Two issues a PERFORM command to call Case Three, and control then returns to Case Two, returns from there to Case One, and finally returns to the Top function. You should not issue a GOTO command from the time the original PERFORM branches out of the Top function until it returns to the Top function.

If, when you code your application, you cannot know every potential run time combination of PERFORM and GOTO branches, it is recommended that you refrain from coding any GOTO commands in your application.

IF

The IF command allows conditional processing depending on how an expression is evaluated.

Syntax: How to Use the IF Command

The syntax of the IF command is

```
IF boolean_expr THEN maint_command [ELSE maint_command]
```

where:

boolean_expr

Is an expression that resolves to a value of true (1) or false (0), and can include stack cells and user-defined fields. For more information about Boolean expressions, see [Expressions Reference](#) on page 21.

Maintain Data handles the format conversion in cases where the expressions have a format mismatch. If the conversion is not possible, a message displays. For additional information, see [Expressions Reference](#) on page 21.

It is highly recommended that parentheses be used when combining expressions. If parentheses are not used, the operators are evaluated in the following order:

1. `**`
2. `* /`
3. `+ -`
4. `LT LE GT GE`
5. `EQ NE`
6. `OMITS CONTAINS`
7. `AND`
8. `OR`

maint_command

You can place any Maintain Data command inside an IF command except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, and MODULE.

Example: Simple Conditional Branching

The following uses an IF command to compare variable values. The function No_ID is performed if the Current Area value of Emp_ID does not equal the value of Emp_ID in StackEmp:

```
IF Emp_ID NE StackEmp(StackEmp.FocIndex).Emp_ID THEN PERFORM No_ID;
   ELSE PERFORM Yes_ID;
```

You might also use an IF command to issue another Maintain Data command. This example causes a COMMIT if there are no errors:

```
IF FocCurrent EQ 0 THEN COMMIT;
```

Example: Using BEGIN to Execute a Block of Conditional Code

This example executes a set of code depending on the value of Department. Additional IF commands could be placed within the BEGIN block of code:

```
IF Department EQ 'MIS' THEN BEGIN
    .
    .
    ENDBEGIN
ELSE IF Department EQ 'MARKETING' THEN BEGIN
    .
    .
    .
```

Example: Nesting IF Commands

IF commands can be nested as deeply as needed, allowing only for memory constraints. The following shows an IF command nested two levels. There is only one IF command after each ELSE:

```
IF Dept EQ 1 THEN TYPE "DEPT EQ 1";
  ELSE IF Dept EQ 2 THEN TYPE "DEPT EQ 2";
  ELSE IF Dept EQ 3 THEN TYPE "DEPT EQ 3";
  ELSE IF Dept EQ 4 THEN TYPE "DEPT EQ 4";
```

This example can be executed more efficiently by issuing the following command:

```
TYPE "DEPT EQ <Dept>";
```

You can also use the BEGIN command to place another IF within a THEN phrase. For example:

```

IF A EQ 1 THEN BEGIN
  IF B EQ 1 THEN BEGIN
    IF C EQ 1 THEN PERFORM C111;
    IF C EQ 2 THEN PERFORM C112;
    IF C EQ 3 THEN PERFORM C113;
  ENDBEGIN
  ELSE IF B EQ 2 THEN BEGIN
    IF C EQ 1 THEN PERFORM C121;
    IF C EQ 2 THEN PERFORM C122;
    IF C EQ 3 THEN PERFORM C123;
  ENDBEGIN
ENDBEGIN
ELSE IF A EQ 2 THEN BEGIN
  IF B EQ 1 THEN BEGIN
    IF C EQ 1 THEN PERFORM C211;
    IF C EQ 2 THEN PERFORM C212;
    IF C EQ 3 THEN PERFORM C213;
  ENDBEGIN
  ELSE IF B EQ 2 THEN BEGIN
    IF C EQ 1 THEN PERFORM C221;
    IF C EQ 2 THEN PERFORM C222;
    IF C EQ 3 THEN PERFORM C223;
  ENDBEGIN
ENDBEGIN
ELSE TYPE "A, B AND C did not have expected values";

```

Coding Conditional COMPUTE Commands

To assign a value to a variable, and the value you assign is conditional upon the truth of an expression, you can use a conditional COMPUTE command. Maintain Data offers you two methods of coding this, using either:

- ❑ An IF command with two COMPUTE commands embedded within it. For example:

```

IF Amount GT 100
  THEN COMPUTE Tfactor/I6 = Amount;
  ELSE COMPUTE Tfactor = Amount * (Factor - Price) / Price;

```

- ❑ A conditional expression within a COMPUTE command. For example:

```

COMPUTE Tfactor/I6 = IF Amount GT 100 THEN Amount
  ELSE Amount * (Factor - Price) / Price;

```

The two methods are equivalent.

INCLUDE

The INCLUDE command inserts segment instances from a transaction source (a stack or the Current Area) into a data source.

When you issue the command, you specify a path running from an anchor segment to a target segment. For each row in the transaction source, INCLUDE searches the data source for matching segment instances and, if none exist, writes the new instances from the transaction source to the data source.

If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor, or provide ancestor segment key values in the source stack. This ensures that INCLUDE can navigate from the root to the first instance of the anchor segment.

Syntax: How to Use the INCLUDE Command

The syntax of the INCLUDE command is

```
[FOR {int|ALL}] INCLUDE path_spec [FROM stack[(row)]] [;]
```

where:

FOR

Is used with ALL or an integer to specify how many stack rows to add to the data source. If FOR is omitted, one stack row will be added.

When you specify FOR, you must also specify FROM to identify a source stack.

int

Is an integer constant or variable that indicates the number of stack rows to add to the data source.

ALL

Specifies that the entire stack is to be added to the data source.

path_spec

Identifies the path to be added to the data source. To identify a path, specify its anchor and target segments. You cannot specify a unique segment as the anchor. If the path contains only one segment, and the anchor and target are identical, simply specify the segment once. For paths with multiple segments, to make the source code clearer to readers, you can also specify segments between the anchor and target.

To add a unique segment instance to a data source, you must explicitly specify the segment in *path_spec*. Otherwise, the unique segment instance will not be added even if it is on the path between the anchor and target segments. This preserves the advantage of assigning space for a unique segment instance only when the instance is needed.

To specify a segment, provide the name of the segment or of a field within the segment.

FROM

Is used to specify a stack containing records to insert. If no stack is specified, data from the Current Area is used.

stack

Is a stack name. Only one stack can be specified.

row

Is a subscript that specifies the first stack row to add to the data source.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Adding Data From Multiple Stack Rows

The following example tries to add the data in rows 2, 3, and 4 of Stkemp into the data source. The stack subscript instructs the system to start in the second row of the stack. The FOR 3 instructs the system to INCLUDE the next three rows.

```
FOR 3 INCLUDE Emp_ID FROM Stkemp(2);
```

Example: Preventing Duplicate Records

You can execute the INCLUDE command after a MATCH command that fails to find a matching record. For example:

```
MATCH Emp_ID FROM Newemp;
ON NOMATCH INCLUDE Emp_ID FROM Newemp;
```

The INCLUDE command can also be issued without a preceding MATCH. In this situation, the key field values are taken from the source stack or Current Area and a MATCH is performed internally. When a set of rows is input without a prior confirmation that it does not already exist in the data source, one or more of the rows in the stack may be rejected. Upon the first rejection, the process stops and the rest of the set is rejected. For all of the rows to be accepted or rejected as a unit, the set should be treated as a logical unit of work and a ROLLBACK issued if the entire set was not accepted. After an INCLUDE, the transaction variable FocError is given a value. If the INCLUDE is successful, FocError is zero (0). If the INCLUDE fails (for example, if the key values already exist in the data source), Maintain Data assigns a non-zero value to FocError, and (if the include was set-based) assigns the value of the row that failed to FocErrorRow. If at COMMIT time there is a concurrency conflict, Maintain Data sets FocCurrent to a non-zero value.

Example: Adding Multiple Segments

This example shows how data is added from two segments in the same path. The data comes from a stack named `EmpInfo` and the entire stack is used. When the `INCLUDE` is complete, the variable `FocError` is checked to see if the `INCLUDE` was successful. If it failed, a general error handling function is called:

```
FOR ALL INCLUDE Emp_ID Dat_Inc FROM EmpInfo;  
IF FocError NE 0 THEN PERFORM Errhandle;
```

Example: Adding Data From the Current Area

The user is prompted for the employee ID and name. The data is included if it does not already exist in the data source. If the data already exists, it is not included, and the variable `FocError` is set to a non-zero value. Since the procedure does not check `FocError`, no error handling takes place and the user does not know whether or not the data is added:

```
NEXT Emp_ID Last_Name First_Name;  
INCLUDE Emp_ID;
```

Reference: Usage Notes for INCLUDE

- If there is a `FOR` prefix, a stack must be mentioned in the `FROM` phrase.
- When an `INCLUDE` command is complete, the variable `FocError` is set. If the `INCLUDE` is successful (the records to be added do not exist in the data source), then `FocError` is set to zero (0). If the records do exist, `FocError` is set to a non-zero value, and (if it is a set-based `INCLUDE`) `FocErrorRow` is set to the number of the row that failed.
- `Maintain Data` requires that data sources to which it writes have unique keys.

Reference: Commands Related to INCLUDE

- COMMIT.** Makes permanent all data source changes since the last `COMMIT`.
- ROLLBACK.** Cancels all data source changes made since the last `COMMIT`.

Data Source Position

A `Maintain Data` procedure always has a position either within a segment or just prior to the first segment instance. If data has been retrieved, the position is the last record successfully retrieved on that segment. If a retrieval operation fails, the data source position remains unchanged.

If an INCLUDE is successful, the data source position is changed to the new record. On the other hand, if the INCLUDE fails, it might be because there is already a record in the data source with the same keys. In this case, the attempted retrieval prior to the INCLUDE is successful, and the position is on that record. Therefore, the position in the data source changes.

Null Values

If you add a segment instance that contains fields for which no data has been provided, and those fields have been defined in the Master File with the MISSING attribute, they are assigned a null value. If those fields have been defined in the Master File without the MISSING attribute, they are assigned a default value of a space for character and date and time fields, or zero (0) for numeric fields.

INFER

Stacks are array variables containing rows and columns. When defining a stack and its structure, provide a name for the stack and a name, format, and order for each of the columns in the stack.

Stacks can be defined in two ways:

- ❑ Performing actual data retrieval with the NEXT command, the stack is defined and populated at the same time. The stack is defined with all the segments that are retrieved. This is convenient when the procedure is processing on the same physical platform as the data source.
- ❑ If the procedure referring to a stack does not retrieve data, you must issue the INFER command to define the structure of the stack. When you issue the command, you specify a data source path. INFER defines the stack with columns corresponding to each field in the specified path. The Master File provides the names and formats of the columns. INFER may only be used to define stack columns that correspond to data source fields. To define user-defined variables, use the COMPUTE command.

A procedure that includes an INFER command must specify the name of the corresponding Master File in the MAINTAIN command, and must have access to the Master File.

Syntax: **How to Use the INFER Command**

The syntax of the INFER command is

```
INFER path_spec INTO stackname [;]
```

where:

path_spec

Identifies the path to be defined for the data source. To identify a path, specify its anchor and target segments. If the path contains only one segment, the anchor and target are identical. Simply specify the segment once. For paths with multiple segments, to make the code clearer to readers, you can also specify segments between the anchor and target.

To specify a segment, provide the name of the segment or of a field within the segment.

stackname

Is the name of the stack.

i

Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: **Inferring Two Stacks**

In the following called procedure, two INFER commands define the EmpClasses and ClassCredits stacks:

```
MAINTAIN FROM EmpClasses INTO ClassCredits
INFER Emp_ID Ed_Hrs Date_Attend Course_Code INTO EmpClasses;
INFER Emp_ID Course_Code Grade Credits INTO ClassCredits;
.
.
.
END
```

Reference: **Commands Related to INFER**

- CALL.** Can be used to call one Maintain Data procedure from another.
- COPY.** Can be used to copy data from one stack to another.
- COMPUTE.** Can be used to define the contents of a stack for non-data source fields.

Defining Non-Data Source Columns

To define stack columns in a procedure for non-data source fields (fields created with the COMPUTE command), you do not need to provide a value for the column. The syntax is:

```
COMPUTE stackname.target_variable/format = ;
```

Note that the equal sign is optional when the COMPUTE is issued solely to establish format.

In the following example, the stack column TempEmp was passed to the called procedure. The COMPUTE is issued in the called procedure to define the variable prior to use:

```
COMPUTE EmpClasses.TempEmp_ID/A9 ;
```

MAINTAIN

The MAINTAIN command marks the beginning of a Maintain Data procedure. You can identify any data sources the procedure will access using the FILE phrase. If the request is to be called from another procedure, you can identify variables to be passed from and to the calling procedure using the FROM and INTO phrases.

Syntax: How to Use the MAINTAIN Command

The syntax of the MAINTAIN command is

```
MAINTAIN [FILE[S] filelist] [FROM varlist] [INTO varlist]
         filelist:filedesc [{AND|,} filedesc ...]
         varlist: {variable} [{variable} ...]
```

where:

MAINTAIN

Identifies the beginning of a Maintain Data request. It must be coded in uppercase letters.

FILE[S]

Indicates that the procedure accesses Master Files. The S can be added to FILE for clarity. The keywords FILE and FILES may be used interchangeably.

You access a Master File when you read or write to a data source, and when you use an INFER command to define the data source columns of a stack. For example, when you redefine a stack that has been passed from a parent procedure.

FROM

Is included if this procedure is called by another procedure, and that procedure passes one or more variables.

INTO

Is included if this procedure is called by another procedure, and this procedure passes one or more variables back to the calling procedure.

filelist

Are the names of the Master Files this procedure accesses.

filedesc

Are the names of the Master Files that describe the data sources that are accessed in the procedure.

AND

Is used to separate Master File names.

,

Is used to separate Master File names.

varlist

Are the variables, both scalar variables and stacks, which are passed to or from this procedure. Multiple variables are separated by blank spaces.

variable

Are the names of the scalar variables or stacks. You can pass any variable, except for those defined as variable-length character (that is, those defined as TX or A0) and those defined using STACK OF.

Reference: Usage Notes for MAINTAIN

- ❑ To access more than one data source, you can specify up to 15 Master Files per MAINTAIN command. If you must access more than 15 data sources, you can call other procedures that can each access an additional 15 data sources.
- ❑ There is a limit of 64 segments per procedure for all referenced data sources, although additional procedures can reference additional segments.

Reference: Commands Related to MAINTAIN

- ❑ **END.** Terminates the execution of a Maintain Data procedure.
- ❑ **CALL.** Is used to call one procedure from another.

Specifying Data Sources With the MAINTAIN Command

The MAINTAIN command does not require any parameters. This means that Maintain Data procedures do not need to access data sources or stacks. You can use a procedure as a subroutine when sharing functions among different procedures, or when certain logic is not executed very frequently. For example, to begin a procedure that does not access any data sources and does not have any stacks as input or output, you simply begin the procedure with the keyword MAINTAIN.

However, the keyword FILE and the name of the Master File are required if you want to access a data source. The following example accesses the Employee data source:

```
MAINTAIN FILE Employee
```

A Maintain Data procedure can access several data sources by naming the corresponding Master Files in the MAINTAIN command:

```
MAINTAIN FILES Employee AND EducFile AND JobFile
```

Calling a Procedure From Another Procedure

You can use the CALL command to pass control to another procedure. When the CALL command is issued, control is passed to the named procedure. Once that procedure is complete, control returns to the item that follows the CALL command in the calling procedure.

Called procedures can also reside on remote Reporting Servers, allowing you to partition the logic of your application between machines.

For information about the CALL command, see [CALL](#) on page 59.

Example: Passing Variables Between Procedures

You can pass stacks and variables between procedures by using FROM and INTO variable lists. In the following example, when the CALL Validate command is reached, control is passed to the procedure named Validate along with the Emps stack. Once Validate is complete, the data in the ValidEmps stack is sent back to the calling procedure. Notice that the calling and called procedures both have the same FROM and INTO stack names. Although this is not required, it is good practice to avoid giving the same stacks different names in different procedures.

The calling procedure contains:

```
MAINTAIN FILE Employee
FOR ALL NEXT Emp_ID INTO Emps;
INFER emp_id into Validemps;
```

```
CALL Validate FROM Emps INTO ValidEmps;  
. . .  
END
```

The called procedure (Validate) contains:

```
MAINTAIN FILE Employee FROM Emps INTO ValidEmps  
. . .  
END
```

MATCH

The MATCH command enables you to identify and retrieve a single segment instance or path instance by key value. You provide the key value using a stack or the Current Area. MATCH finds the first instance in the segment chain that has that key.

You specify which path to retrieve by identifying its anchor and target segments. If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor. This enables MATCH to navigate from the root to the anchor segment instance.

The command always matches on the full key. To match on a partial key, use the NEXT command and identify the value of the partial key in the WHERE phrase of the command.

If the data source has been defined without a key, you can retrieve a segment instance or path using the NEXT command, and identify the desired instance using the WHERE phrase of the command.

Syntax: How to Use the MATCH Command

The syntax of the MATCH command is

```
MATCH path_spec [FROM stack[(row)]] [INTO stack[(row)]] [;]
```

where:

path_spec

Identifies the path to be read from the data source. To identify a path, specify its anchor and target segments. If the path contains only one segment, the anchor and target are identical. Specify the segment once. For paths with multiple segments, to make the code clearer to readers, you can also specify segments between the anchor and target.

To specify a segment, provide the name of the segment or of a field within the segment.

FROM

Is used to specify a stack containing a key value on which to match. If you omit this, Maintain Data uses a value in the Current Area. In either case, the columns containing the key value must have the same names as the corresponding key fields in the data source.

INTO

Is used to specify the stack that the data source values are to be placed into. Values retrieved by MATCH are placed into the Current Area when an INTO stack is not supplied.

stack

Is a stack name. Only one stack can be specified for each FROM or INTO phrase. The stack name should have a subscript specifying which row is to be used. If a stack is not specified, the values retrieved by the MATCH go into the Current Area.

row

Is a subscript that specifies which row is used. The first row in the stack is matched against the data source if the FROM stack does not have a subscript. The data is placed in the first row in the stack if the INTO stack does not have a subscript.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Matching Keys in the Employee Data Source

The following example performs a MATCH on the key field in the PayInfo segment. It gets the value for Pay_Date from the Pay_Date field, which is in the Current Area. After the match is found, all of the field values in the PayInfo segment are copied from the data source into the Current Area:

```
MATCH Pay_Date;
```

The next example shows a MATCH on the key in the EmplInfo segment. It gets the value for Emp_ID from the Emp_ID column in the Cnt row of the Stackemp stack. After the match is found, all of the fields in the EmplInfo segment are copied into the Current Area:

```
MATCH Emp_ID FROM Stackemp(Cnt);
```

The last example is the same as the previous example except that an output stack is mentioned. The only difference in execution is that after the match is found, all of the fields in the EmplInfo segment are copied into a specific row of a stack rather than into the Current Area:

```
MATCH Emp_ID FROM Stackemp(Cnt) INTO Empout(Cnt);
```

Reference: Commands Related to MATCH

- ❑ **NEXT.** Starts at the current position and moves forward through the data source. NEXT can retrieve data from one or more records.
- ❑ **REPOSITION.** Changes the data source position to be at the beginning of the chain.

How the MATCH Command Works

When a MATCH command is issued, Maintain Data tries to retrieve a corresponding record from the data source. If there is no corresponding value and an ON NOMATCH command follows, the command is executed.

The MATCH command looks through the entire segment to find a match. The NEXT command with a WHERE qualifier also locates a data source record, but does it as a forward search. That is to say, it starts at its current position and moves forward. It is not an exhaustive search unless positioned at the start of a segment. This can always be done with the REPOSITION command. A MATCH is equivalent to a REPOSITION on the segment followed by a NEXT command with a WHERE phrase specifying the key. If any type of test other than the equality test that the MATCH command provides is needed, the NEXT command should be used.

MODULE

The MODULE command accesses a source code library so the current procedure can use the class definitions of the library and Maintain Data functions. A library is a nonexecutable procedure, and is implemented as a component called an import module.

Syntax: How to Use the MODULE Command

The MODULE command must immediately follow the MAINTAIN command. The syntax of the MODULE command is

```
MODULE IMPORT (library_name [,library_name] ... );
```

where:

library_name

Is the name of the library to import as a source code library. Specify its file name without an extension. The file must reside in the path defined by the EDASYNR environment variable.

If a library is specified multiple times in a MODULE command, Maintain Data will include the library only once in order to avoid a loop.

Reference: Commands Related to MODULE

- ❑ **DESCRIBE.** Defines classes. You can use DESCRIBE to include classes in a library.
- ❑ **CASE.** Defines a function. You can use CASE to include functions in a library.

What You Can and Cannot Include in a Library

You can include most Maintain Data language commands and structures in a library. However, there are some special opportunities and restrictions of which you should take note:

- ❑ **Other libraries.** You can place one library within another, and can nest libraries to any depth. For example, to nest library B within library A, issue a MODULE IMPORT B command within library A.

If a given library is specified more than once in a series of nested libraries, Maintain Data will only include the library once in order to avoid a loop.

- ❑ **Top function.** Because a library is a nonexecutable procedure, it has no Top function.
- ❑ **Forms.** A library cannot contain forms.
- ❑ **Data sources.** A library cannot refer to data sources. For example, it cannot contain data source commands (such as NEXT and INCLUDE) and cannot refer to data source stacks.

NEXT

The NEXT command selects and reads segment instances from a data source. You can use NEXT to read an entire set of records at a time, or just a single segment instance. You can select segments by field value or sequentially.

You specify a path running from an anchor segment to a target segment. NEXT reads all the fields from the anchor through the target, and (if the anchor segment is not the root) all the keys of the ancestor segments of the anchor. It copies what it has read to the stack that you specify or, if you omit a stack name, to the Current Area.

If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor. This enables NEXT to navigate from the root to the current instance of the anchor segment.

In each segment that it reads, NEXT works its way forward through the segment chain. When no more records are available, the NONEXT condition arises and no more records are retrieved unless the procedure issues a REPOSITION command. REPOSITION causes a reposition to just prior to the beginning of the segment chain. If you are familiar with the SQL language, the NEXT command acts as a combination of the SQL commands SELECT and FETCH, and allows you to use the structure of the data source to your advantage when retrieving data.

Syntax: **How to Use the NEXT Command**

The syntax of the NEXT command is

```
[FOR {int|ALL}] NEXT path [INTO stack[(row)]] [WHERE where_expression1  
[AND where_expression2 ...]] [;]
```

where:

FOR

Is a prefix that is used with *int* or ALL to specify how many data source records are to be retrieved. If FOR is not specified, NEXT works like FOR 1 and the next record is retrieved. If the FOR phrase is used, the INTO phrase must also be used.

int

Is an integer constant or variable that specifies the number of data source records that are retrieved from the data source. Retrieval starts at the current position in the data source.

ALL

Specifies that starting at the current data source position, all data source segments referred to in the field list are examined.

path

Identifies the path to be read from the data source. To identify a path, specify its anchor and target segments. If the path contains only one segment, the anchor and target are identical, simply specify the segment once. For paths with multiple segments, to make the code clearer to readers, you can also specify segments between the anchor and target.

To specify a segment, provide the name of the segment or of a field within the segment.

INTO

Is used with a stack name to specify the name of the stack into which the data source records are copied.

stack

Is the name of the stack that the data source values are placed into. Only one stack can be specified.

row

Is a subscript that specifies in which row of the stack the data source values are placed. If no subscript is provided, the data is placed in the stack starting with the first row.

where_expression1, where_expression2

Is any valid NEXT WHERE expression. You can use any valid relational expression, described in [Relational Expressions](#) on page 44. NEXT can also use some enhanced screening conditions not available in other situations. For more information, see [Using Selection Logic to Retrieve Rows](#) on page 123.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Reference: Usage Notes for NEXT

- ❑ If an INTO stack is specified, and that stack already exists, new rows are added starting at the row specified. If no stack row number is specified, then data is added starting at the first row. In either case, it is possible that some existing rows may be written over. If a NEXT command causes only some of the rows in a stack to be overwritten, the rest of the stack remains intact. If the subscript provided on the INTO stack is past the end of the existing stack, the intervening rows are initialized to spaces, zeroes (0), or nulls (missing values), as appropriate. If the new stack overwrites some of the rows of the existing stack, only those rows are affected. The rest of the stack remains intact.
- ❑ If no FOR prefix is used and no stack name is supplied, the values retrieved by the NEXT command go into the Current Area.

Reference: Commands Related to NEXT

- ❑ **REPOSITION.** Changes the data source position to be at the beginning of the chain.
- ❑ **MATCH.** Searches the entire segment for a matching field value. It retrieves an exact match in the data source.

Copying Data Between Data Sources

You can use the NEXT command to copy data between data sources. It is helpful to copy data between data sources when transaction data is gathered by one application and must be stored for use by another application. It is also helpful when the transaction data is to be applied to the data source at a later time or in a batch environment.

Example: Copying Data to the Movies Data Source

For example, assume that you want to copy data from a fixed-format data source named FilmData into an App Studio data source named Movies. You describe FilmData using the following Master File:

```
FILENAME=FILMDATA,  SUFFIX=FIX
SEGNAME=MOVINFO,   SEGTYPE=S0
  FIELDNAME=MOVIECODE,  ALIAS=MCOD,  USAGE=A6,  ACTUAL=A6,$
  FIELDNAME=TITLE,     ALIAS=MTL,   USAGE=A39, ACTUAL=A39,$
  FIELDNAME=CATEGORY,  ALIAS=CLASS, USAGE=A8,  ACTUAL=A8,$
  FIELDNAME=DIRECTOR,  ALIAS=DIR,  USAGE=A17, ACTUAL=A17,$
  FIELDNAME=RATING,    ALIAS=RTG,  USAGE=A4,  ACTUAL=A4,$
  FIELDNAME=RELDATE,   ALIAS=RDAT,  USAGE=YMD, ACTUAL=A6,$
  FIELDNAME=WHOLESALEPR, ALIAS=WPRC,  USAGE=F6.2, ACTUAL=A6,$
  FIELDNAME=LISTPR,    ALIAS=LPRC,  USAGE=F6.2, ACTUAL=A6,$
  FIELDNAME=COPIES,    ALIAS=NOC,   USAGE=I3,  ACTUAL=A3,$
```

The fields in FilmData have been named identically to those in Movies to establish the correspondence between them in the INCLUDE command that writes the data to Movies.

You can read FilmData into Movies using the following procedure:

```
MAINTAIN FILE Movies AND FilmData
FOR ALL NEXT FilmData.MovieCode INTO FilmStack;
FOR ALL INCLUDE Movies.MovieCode FROM FilmStack;
END
```

All field names in the procedure are qualified to distinguish between identically-named fields in the input data source (FilmData) and the output data source (Movies).

Loading Multi-Path Transaction Data

To load data from a transaction data source into multiple paths of a data source, you should process each path independently. Use one pair of NEXT and INCLUDE commands per path.

For example, assume that you have a transaction data source named TranFile whose structure is identical to that of the VideoTrk data source.

To load the transaction data from both paths of TranFile into both paths of VideoTrk, you could use the following procedure:

```

MAINTAIN FILES TranFile AND VideoTrk
FOR ALL NEXT TranFile.CustID TranFile.ProdCode INTO ProdStack;
REPOSITION CustID;
FOR ALL NEXT TranFile.CustID TranFile.MovieCode INTO MovieStack;
FOR ALL INCLUDE VideoTrk.CustID VideoTrk.ProdCode FROM ProdStack;
FOR ALL INCLUDE VideoTrk.CustID VideoTrk.MovieCode FROM MovieStack;
END

```

Alternatively, if you choose to store each path of transaction data in a separate single-segment transaction data source, the same principles apply. For example, if the two paths of TranFile are stored separately in transaction data sources TranProd and TranMove, the previous procedure would change as highlighted below:

```

MAINTAIN FILES TranProd AND TranMove AND VideoTrk
FOR ALL NEXT TranProd.CustID INTO ProdStack;
FOR ALL NEXT TranMove.CustID
  INTO MovieStack;
FOR ALL INCLUDE VideoTrk.CustID VideoTrk.ProdCode FROM ProdStack;
FOR ALL INCLUDE VideoTrk.CustID VideoTrk.MovieCode FROM MovieStack;
END

```

Retrieving Multiple Rows: The FOR Phrase

The FOR phrase is used to specify the number of data source records that are to be retrieved. As an example, if FOR 10 is used, ten records are retrieved. A subsequent FOR 10 retrieves the next ten records starting from the last position. If an attempt to retrieve ten records only returns seven because the end of the chain is reached, the command retrieves seven records, and the ON NONEXT condition is raised.

The following retrieves the next ten instances of the EmplInfo segment and places them into Stackemp:

```
FOR 10 NEXT Emp_ID INTO Stackemp;
```

Using Selection Logic to Retrieve Rows

When you are retrieving rows using the NEXT command, you have the option to restrict the rows you retrieve using the WHERE clause. The syntax for this option is

```

WHERE operand1 comparison_op1 operand2
[AND operand3 comparison_op1 operand4 ...]

```

where:

```
operand1, operand2, operand3, operand4, ...
```

Are operands. In each NEXT WHERE expression, one operand must be a data source field, and one must be a valid Maintain Data expression that does not refer to a data source field.

For more information about Maintain Data expressions, see [Expressions Reference](#) on page 21.

comparison_op1, comparison_op2, ...

Can be any of the comparison operators listed in [Logical Operators](#) on page 45 or any of the comparison operators listed in [Comparison Operators](#) on page 124. Some comparison operators may be listed in both places. This means that they can be used in a WHERE clause in an enhanced way.

The following example retrieves every instance of the EmpInfo segment that has a department value of MIS:

```
FOR ALL NEXT Emp_ID INTO EmpStack WHERE Department EQ 'MIS';
```

Literals can be enclosed in either single quotation marks (') or double quotation marks ("). For example, the following produces exactly the same results as the last example:

```
FOR ALL NEXT Emp_ID INTO EmpStack WHERE Department EQ "MIS";
```

The ability to use either single quotation marks (') or double quotation marks (") provides the added flexibility of being able to use either single quotation marks (') or double quotation marks (") in text. For example:

```
NEXT Emp_ID WHERE Last_Name EQ "O'HARA";  
NEXT Product WHERE Descr CONTAINS 'TEST';
```

This example starts at the beginning of the segment chain and searches for all employees that are in the MIS department. All retrieved segment instances are copied into a stack:

```
REPOSITION Emp_ID;  
FOR ALL NEXT Emp_ID INTO Misdept WHERE Department EQ 'MIS';
```

After FOR ALL NEXT is processed, you are positioned at the end of the segment chain. Therefore, before issuing an additional NEXT command on the same segment chain, you should issue a REPOSITION command to be positioned prior to the first instance in the segment chain.

Reference: Comparison Operators

IS, EQ, NE, IS_NOT

Select data source values using wildcard characters (you embed the wildcards in a character constant in the non-data source operand). You can use dollar sign wildcards (\$) throughout the constant to signify that any character is acceptable in the corresponding position of the data source value.

To allow any value of any length at the end of the data source value, you can combine a dollar sign wildcard with an asterisk (\$*) at the end of the constant.

For example:

```
WHERE ZipCode IS '112$$'
```

CONTAINS, OMTS

Select data source values that contain or omit a character string stored in a variable.

For example, the following returns all data where the word BANK is part of the bank name:

```
COMPUTE name/A4 = 'BANK';

FOR ALL NEXT bank_code
bank_name into stackname

WHERE bank_name CONTAINS name;
```

The following returns all data where the bank name does not include the word BANK:

```
COMPUTE name/A4 = 'BANK';

FOR ALL NEXT bank_code
bank_name into stackname

WHERE bank_name OMTS name;
```

EXCEEDS

Selects data source values that are greater than a numeric value.

For example:

```
WHERE TOTAL Curr_sal EXCEEDS 110000
```

IN (*list*), NOT_IN (*list*)

Select data source values that are in or not in a list. IN and NOT_IN can be used with all data types.

For example, the following returns all data where the bank name is not in the list:

```
FOR ALL NEXT emp_id bank_name INTO stackname

WHERE bank_name NOT_IN
('ASSOCIATED BANK', CITIBANK)
```

EQ_MASK, NE_MASK

Select data source values that match or do not match a mask.

Use the \$ sign to replace each letter in the value. Masks can only be used with alphanumeric data. The masked value may be hard coded or a variable.

For example, the following returns all data where the bank code starts with AAA and has any character at the end:

```
COMPUTE code/A4='AAA$';

FOR ALL NEXT bank_name
INTO stackname

WHERE bank_code EQ_Mask code;
```

The following returns all data where the bank code does not match the mask:

```
COMPUTE code/A4='AAA$';

FOR ALL NEXT bank_name
INTO Stackname

WHERE bank_code NE_Mask code;
```

Using NEXT After a MATCH

NEXT can also be used in conjunction with the MATCH command. This example issues a MATCH for employee ID. If there is not a match, a message displays. If there is a match, all the instances of the child segment for that employee are retrieved and placed in the stack Stackemp. The NEXT command can be coded as part of an ON MATCH condition, but it is not required, as the NEXT will only retrieve data based on the current position of higher-level segments.

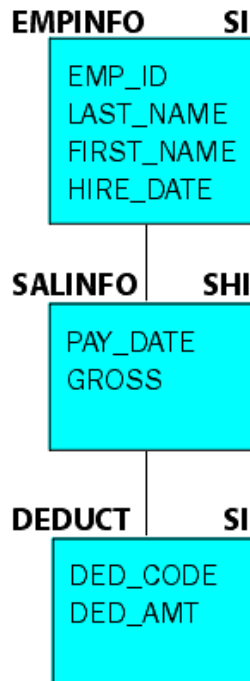
```
MATCH Emp_ID
ON NOMATCH BEGIN
  TYPE "The employee ID is not in the data source.";
  GOTO Getmore;
ENDBEGIN
FOR ALL NEXT Dat_Inc INTO Stackemp;
```

Using NEXT for Data Source Navigation: Overview

The segments that NEXT operates on are determined by the fields mentioned in the NEXT command. The list of fields is used to determine the anchor segment (the segment closest to the root) and the target segment (the segment furthest from the root). Every segment starting with the anchor and ending with the target make up the scope of the NEXT command, including segments not mentioned in the NEXT command. Both the target and the anchor must be in one data source path.

NEXT does not retrieve outside the scope of the anchor and target segment. All segments not referenced remain constant, which is why NEXT can be used as a get next within parent (GNP).

As an example, look at a partial view of the Employee data source:



If a NEXT command has SallInfo as the anchor segment and the target is the Deduct segment, it also needs to retrieve data for the EmplInfo segment, which is the parent of the SallInfo segment based on its current position. The position for the EmplInfo segment can be established by either a prior MATCH or NEXT command. If no position has been established for the EmplInfo segment, an error occurs.

You can use the NEXT command for:

- [Data Source Navigation: Using NEXT With One Segment](#) on page 128.
- [Data Source Navigation: Using NEXT With Multiple Segments](#) on page 129.
- [Data Source Navigation: Using NEXT Following NEXT or MATCH](#) on page 131.

Data Source Navigation: Using NEXT With One Segment

If a NEXT references only one segment and has no WHERE phrase or FOR prefix, it always moves forward one instance within that segment. If the segment is not the root, all parent segments must have a position in the data source and only those instances descending from those parents are examined and potentially retrieved. The NEXT command starts at the current position within the segment, and each time the command is encountered, it moves forward one instance. If a prior position has not been established within the segment (no prior NEXT, MATCH, or REPOSITION command has been issued), the NEXT retrieves the first instance.

The following command references the root segment, so there is no parent segment in which to have a position:

```
NEXT Emp_ID;
```

The following command refers to a child segment, so the parents to this segment must already have a position and that position does not change during the NEXT operation:

```
NEXT Pay_Date;
```

If the NEXT command uses the FOR prefix, it works the same as described above, but rather than moving forward only one data source instance, it moves forward as many rows as the FOR specifies. The following retrieves the next three instances of the EmplInfo segment:

```
FOR 3 NEXT Emp_ID INTO Stemp;
```

If a FOR prefix is used, an INTO stack must be specified. However, an INTO stack can be specified without the FOR prefix.

If a WHERE phrase is specified and there is no FOR prefix, the NEXT moves forward as many times as necessary to retrieve one row that satisfies the selection criteria specified in the WHERE phrase. The following retrieves the next employee in the MIS department:

```
NEXT Emp_ID WHERE Department EQ 'MIS';
```

If the NEXT command does not have an INTO stack name, the output of the NEXT (the value of all of the fields in the segment instance) goes into the Current Area. If an INTO stack is specified, the output goes into the stack named in the command. If more than one row is retrieved by using a FOR prefix, the number of rows specified in the FOR are placed in the stack. If the INTO stack specifies a row number (for example, INTO Mystack(10)), then the rows are added to the stack starting with that row number. If the INTO stack does not specify a row number, the rows are added to the stack starting at the first row.

The following retrieves all of the fields from the next instance in the segment that Emp_ID is in and places the output into the first row of the Stemp stack:


```
NEXT Emp_ID INTO Stemp;
```

If the NEXT command has both a WHERE phrase and a FOR prefix, it moves forward as many times as necessary to retrieve the number of rows specified in the FOR phrase that satisfies the selection criteria specified in the WHERE phrase. The following retrieves the next three employees in the MIS department and places the output into the stack called Stemp:

```
FOR 3 NEXT Emp_ID INTO Stemp WHERE Department EQ 'MIS';
```

If there were not as many rows retrieved as you specified in the FOR prefix, you can determine how many rows were actually retrieved by checking the FocCount variable of the target stack.

Data Source Navigation: Using NEXT With Multiple Segments

If a NEXT command references more than one segment, each time the command is executed it moves forward within the target (the lowest-level child segment). Once the target no longer has any more instances, the next NEXT moves forward on the parent of the target and repositions itself at the beginning of the chain of the child. In the following example, the REPOSITION command changes the position of EmplInfo to the beginning of the data source (EmplInfo is in the root). The first NEXT command finds the first instance of both segments. When the second NEXT is executed, what happens depends on whether there is another instance of the SallInfo segment, because the NEXT command does not retrieve short path instances (that is, it does not retrieve path instances that are missing descendant segments). If there is another instance, the second NEXT moves forward one instance in the SallInfo segment. If there is only one instance in the SallInfo segment for the employee retrieved in the first NEXT, the second NEXT moves forward one instance in the EmplInfo segment. When this happens, the SallInfo segment is positioned at the beginning of the chain and the first SallInfo instance is retrieved. If there is no instance of SallInfo, the NEXT command retrieves the next record that has a SallInfo segment instance.

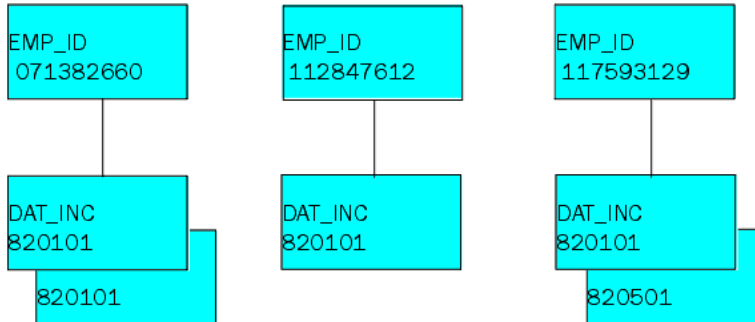
```
REPOSITION Emp_ID;
NEXT Emp_ID Pay_Date;
NEXT Emp_ID Pay_Date;
```

When there is a possibility of short paths, and the intention is to retrieve the data from the parent even if there is no data for the child, NEXT should be used on one segment at a time, as described in [Data Source Navigation: Using NEXT Following NEXT or MATCH](#) on page 131. If a NEXT command uses the FOR *n* prefix, it works the same as described above, but rather than moving forward only one data source instance, it moves forward as many records as are required to retrieve the number specified in the FOR prefix.

For instance, the following command retrieves the next five instances of the EmplInfo and SallInfo segments and places the output into the Stemp stack. The five records may or may not all have the same EmplInfo segment:

```
FOR 5 NEXT Emp_ID Dat_Inc INTO Stemp;
```

If the data source is populated as follows,



all of the fields from the following segment instances are added to the stack:

1. EMP_ID 071382660, DAT_INC 820101
2. EMP_ID 071382660, DAT_INC 810101
3. EMP_ID 112847612, DAT_INC 820101
4. EMP_ID 117593129, DAT_INC 820601
5. EMP_ID 117593129, DAT_INC 820501

If a WHERE phrase is specified, the NEXT moves forward as many times as necessary to retrieve one record that satisfies the selection criteria specified in the WHERE phrase. For example, the following retrieves the next record where the child segment has the field Gross greater than 1,000:

```
NEXT Emp_ID Pay_Date WHERE Gross GT 1000;
```

If both a WHERE phrase and a FOR prefix are specified, the NEXT moves forward as many times as necessary to retrieve the number specified in the FOR prefix that satisfies the selection criteria specified in the WHERE phrase. For instance, the following retrieves all of the records where the Gross field is greater than 1,000. As stated above, if more than one segment is mentioned and there is a FOR prefix, the data retrieved may come from more than one employee:

```
FOR ALL NEXT Emp_ID Pay_Date INTO Stemp WHERE Gross GT 1000;
```

If the NEXT command does not have an INTO stack name provided, the output of the NEXT is copied into the Current Area. If an INTO stack is specified, the output is copied into the stack named in the command. The number of records retrieved is the number that is placed in the stack. If the INTO stack specifies a row number (for example, INTO Mystack(10)) then the records are added to the stack starting at the row number. If the INTO stack does not specify a row number, the rows are added to the stack starting with the first row in the stack. If data already exists in any of the rows, those rows are cleared and replaced with the new values.

If the NEXT command can potentially retrieve more than one record (the FOR prefix is used), an INTO stack must be specified. If no stack is provided, a message displays and the procedure terminates.

Data Source Navigation: Using NEXT Following NEXT or MATCH

In order to use NEXT through several segments, specify all the segments in one NEXT command or use several NEXT commands. If all of the segments are placed into one NEXT command, there is no way to know when data is retrieved from a parent segment and when it is retrieved from a child. To have control over when each segment is retrieved, each segment should have a NEXT command of its own. In this way, the first NEXT establishes the position for the second NEXT.

A NEXT command following a MATCH command works in a similar way. The first command (MATCH) establishes the data source position.

In the following example, the REPOSITION command places the position in the EmplInfo segment and all of its children to the beginning of the chain. Both NEXT commands move forward to the first instance in the appropriate segment:

```
REPOSITION Emp_ID;
NEXT Emp_ID;
NEXT Pay_Date;
```

If one of the NEXT commands uses the FOR prefix, it works the same as described above, but rather than moving forward only one segment instance, NEXT moves forward however many records the FOR specifies. For example, the following retrieves the first instance in the EmplInfo segment and the next three instances of the SallInfo segment. All three records are for only one employee because the first NEXT establishes the position:

```
REPOSITION Emp_ID;
NEXT Emp_ID;
FOR 3 NEXT Pay_Date INTO Stemp;
```

After this code is executed, the stack contains data from the following segments:

1. Emp_ID instance 1 and Pay_Date instance 1

2. Emp_ID instance 1 and Pay_Date instance 2
3. Emp_ID instance 1 and Pay_Date instance 3

Every NEXT command that uses a FOR prefix does so independent of any other NEXT command. If there are two NEXT commands, the first executes. When it is complete, the position is the last instance retrieved. The second NEXT command then executes and retrieves data from within the parent established by the first NEXT. In the following example, the first NEXT retrieves the first two instances from the EmplInfo segment and places the instances into the stack. The second NEXT retrieves the next three instances of the SallInfo segment. Note its parent instance is the second EmplInfo segment instance. The stack variable FocCount indicates the number of rows currently in the stack. The prefix Stemp is needed to indicate the stack.

```
STACK CLEAR Stemp;  
REPOSITION Emp_ID;  
FOR 2 NEXT Emp_ID INTO Stemp(1);  
FOR 3 NEXT Pay_Date INTO Stemp(Stemp.FocCount);
```

The stack contains data from the following segments after the first NEXT is executed:

1. Emp_ID instance 1
2. Emp_ID instance 2

The stack contains data from the following segments after the second NEXT is executed:

1. Emp_ID instance 1
2. Emp_ID instance 2 and Pay_Date instance 1
3. Emp_ID instance 2 and Pay_Date instance 2
4. Emp_ID instance 2 and Pay_Date instance 3

The row in the INTO stack that the output is placed in is specified by supplying the row number after the stack name. When two NEXT commands are used in a row for the same stack, care must be taken to ensure that data is written to the appropriate row in the stack. If a stack row number is not specified for the second NEXT command, data is placed into the last row written to by the first NEXT, and existing data is overwritten. In order to place data in a different row, a row number or an expression to calculate the row number can be used. For example, the second NEXT command specifies the row after the last row by adding one to the variable FocCount:

```
FOR 2 NEXT Emp_ID INTO Stemp(1);  
FOR 3 NEXT Pay_Date INTO Stemp(Stemp.FocCount+1);
```

The stack now appears as follows. Notice that there is a new row 2:

1. Emp_ID instance 1

2. Emp_ID instance 2
3. Emp_ID instance 2 and Pay_Date instance 1
4. Emp_ID instance 2 and Pay_Date instance 2
5. Emp_ID instance 2 and Pay_Date instance 3

If a WHERE phrase is specified, the NEXT moves forward as many times as necessary to retrieve one record that satisfies the selection criteria specified in the WHERE phrase. For instance, the following retrieves the next record where the Gross field of the child segment is greater than 1,000. Like the previous example, the data retrieved is only for the employee that the first NEXT retrieves:

```
NEXT Emp_ID;
NEXT Pay_Date WHERE Gross GT 1000;
```

If both a FOR prefix and a WHERE phrase are specified, the NEXT moves forward as many times as necessary to retrieve the number of records specified in the FOR prefix that satisfy the selection criteria specified in the WHERE phrase.

For example, the following syntax retrieves the next three records where the Gross field of the child segment is greater than 1,000. As above, the data retrieved is only for the employee that the first NEXT retrieves:

```
NEXT Emp_ID;
FOR 3 NEXT Pay_Date INTO Stemp WHERE Gross GT 1000;
```

Unique Segments

Maintain Data allows separate segments to be joined in a one-to-one relation (among other ways). Unique segments are indicated by specifying a SEGTYPE of U, KU, or DKU in the Master File, or by issuing a JOIN command. In a NEXT command, you retrieve a unique segment by specifying a field from the segment in the field list of the command. You cannot specify the unique segment as an anchor segment.

If an attempt is made to retrieve data from a unique segment and the segment does not exist, the fields are treated as if they are fields in the parent segment. This means that the returned data is spaces, zeroes (0), and/or nulls (missing values), depending on how the segment is defined. In addition, the answer set contains as many rows as the parent of the unique segment. If an UPDATE or a DELETE command subsequently uses the data in the stack and the unique segment does not exist, it is not an error, because unique segments are treated as if the fields are fields in the parent. If an INCLUDE is issued, the data source is not updated.

ON MATCH

The ON MATCH command defines the action to take if the prior MATCH command succeeds (if it is able to retrieve the specified segment instance). There can be intervening commands between the MATCH and ON MATCH commands, and they can be in separate functions.

You should query the FocFetch system variable in place of issuing the ON MATCH command. FocFetch accomplishes the same thing more efficiently. For more information, see [FocFetch](#) on page 99.

Syntax: How to Use the ON MATCH Command

The syntax of the ON MATCH command is

```
ON MATCH command
```

where:

```
command
```

Is the action that is taken when the prior MATCH command succeeds.

You can specify any Maintain Data command except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, MODULE, and another ON command.

Example: Using On MATCH

The following example displays a message stating that there was a MATCH:

```
MATCH Emp_ID;  
ON MATCH TYPE "Employee was found in the data source";
```

This example shows an UPDATE that is performed after a MATCH occurs:

```
MATCH Emp_ID;  
ON MATCH UPDATE Salary FROM SalStack;
```

The following shows several commands being executed after a MATCH:

```
MATCH Emp_ID;  
ON MATCH BEGIN  
  TYPE "Employee was found in the data source";  
  UPDATE Salary FROM Salstack;  
  PERFORM Jobs;  
ENDBEGIN
```

ON NEXT

The ON NEXT command defines the action to take if the prior NEXT command succeeds (if it is able to retrieve *all* of the specified records). There can be intervening commands between the NEXT and ON NEXT commands, and they can be in separate functions.

It is recommended that you query the FocFetch system variable in place of issuing the ON NEXT command. FocFetch accomplishes the same thing more efficiently. For more information, see [FocFetch](#) on page 99.

Syntax: How to Use the ON NEXT Command

The syntax of the ON NEXT command is

```
ON NEXT command
```

where:

```
command
```

Is the action that is taken when NEXT is successful.

You can specify any Maintain Data command except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, MODULE, and another ON command.

Example: Using ON NEXT

The first example displays a message stating that the NEXT was successful:

```
NEXT Emp_ID;
ON NEXT TYPE "Was able to NEXT another employee";
```

This example computes a five percent increase for the next employee in the data source:

```
NEXT Emp_ID;
ON NEXT COMPUTE NewSal/D12.2 = Curr_Sal * 1.05;
```

The following example shows several commands that are executed after a NEXT:

```
ON NEXT BEGIN
  TYPE "Was able to NEXT another employee";
  COMPUTE NewSal/D12.2 = Curr_Sal * 1.05;
ENDBEGIN
```

ON NOMATCH

The ON NOMATCH command defines the action to take if the prior MATCH command fails (if it is unable to retrieve the specified segment instance). There can be intervening commands between the MATCH and ON NOMATCH commands, and they can be in separate functions.

It is recommended that you query the FocFetch system variable in place of issuing the ON NOMATCH command. FocFetch accomplishes the same thing more efficiently. For more information, see *FocFetch* on page 99.

Syntax: **How to Use the ON NOMATCH Command**

The syntax of the ON NOMATCH command is

```
ON NOMATCH command
```

where:

command

Is the action that is taken when the prior MATCH command fails.

You can specify any Maintain Data command except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, MODULE, and another ON command.

Example: **Using ON NOMATCH**

The first example displays a message stating that the MATCH was unsuccessful:

```
MATCH Emp_ID;
ON NOMATCH TYPE "Employee was not found in the data source";
```

This example shows an INCLUDE of a row from the Emp stack:

```
MATCH Emp_ID FROM Emp(Cnt);
ON NOMATCH INCLUDE Emp_ID FROM Emp(Cnt);
```

The following example shows several commands that are executed after a MATCH command fails:

```
MATCH Emp_ID;
ON NOMATCH BEGIN
    TYPE "Employee was not found in the data source";
    INCLUDE Emp_ID;
    PERFORM Cleanup;
ENDBEGIN
```

ON NONEXT

The ON NONEXT command defines the action to take if the prior NEXT command fails (if it is unable to retrieve *all* of the specified records). There can be intervening commands between the NEXT and ON NONEXT commands, and they can be in separate functions.

For example, when the following NEXT command is executed

```
FOR 10 NEXT Emp_ID INTO Stkemp;
```


only eight employees are left in the data source, so only eight records are retrieved, raising the ON NONEXT condition.

It is recommended that you query the FocFetch system variable in place of issuing the ON NONEXT command. FocFetch accomplishes the same thing more efficiently. For more information, see [FocFetch](#) on page 99.

Syntax: **How to Use the ON NONEXT Command**

The syntax of the ON NONEXT command is

```
ON NONEXT command
```

where:

command

Is the action that is taken when NEXT fails.

You can specify any Maintain Data command except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, MODULE, and another ON command.

Example: **Using ON NONEXT**

The first example displays a message stating that the NEXT was unsuccessful:

```
NEXT Emp_ID;
ON NONEXT TYPE "There are no more employees";
```

If all of the employees have been processed, the program is exited:

```
NEXT Emp_ID;
ON NONEXT GOTO EXIT;
```

The following example shows several commands being executed after a NEXT fails:

```
ON NONEXT BEGIN
  TYPE "There are no more employees in the data source";
  PERFORM Wrapup;
ENDBEGIN
```

PERFORM

You can use the PERFORM command to pass control to a Maintain Data function. Once that function is executed, control returns to the command immediately following the PERFORM.

Syntax: **How to Use the PERFORM Command**

The syntax of the PERFORM command is

```
PERFORM functionname [()] [;]
```

where:

functionname

Specifies the name of the function to perform.

()

Is optional. If you omit the word PERFORM and only use the function name, parentheses are required.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

For example, to perform the function named NextSet, issue the command:

```
PERFORM NextSet;
```

Reference: Commands Related to PERFORM

- ❑ **CASE/ENDCASE.** Defines a Maintain Data function.
- ❑ **GOTO.** Transfers control to another function or to the end of the current function.

Using PERFORM to Call Maintain Data Functions

When you call a function as a separate statement (that is, outside of a larger expression), if the preceding command can take an optional semicolon (;) terminator but was coded without one, you must call the function in a COMPUTE or PERFORM command. You can use PERFORM for Maintain Data functions only, though not for Maintain Data functions that return a value.

For example, in the following source code, the NEXT command does not end with a semicolon (;), so the function that follows it must be called in a PERFORM command:

```
NEXT CustID INTO CustStack
PERFORM VerifyCustID();
```

However, in all other situations, you can call functions directly, without a PERFORM command. For example, in the following source code, the NEXT command ends with a semicolon (;), so the function that follows it can be called without a PERFORM command:

```
NEXT CustID INTO CustStack;
VerifyCustID();
```

Note: When calling a function without using a PERFORM command, you must include parentheses.

For more information about terminating commands with a semicolon (;), see [Terminating Command Syntax](#) on page 17.

Using PERFORM With Data Source Commands

A PERFORM can be executed in a MATCH command following an ON MATCH or ON NOMATCH command, or in NEXT following ON NEXT or ON NONEXT. In the following example, the function NotHere is performed after a NOMATCH condition occurs:

```
ON NOMATCH PERFORM NotHere;
```

Nesting PERFORM Commands

PERFORM commands can branch to functions containing other PERFORM commands. As each ENDCASE command is encountered, control returns to the command after the most recently executed PERFORM command. In this manner, control eventually returns to the original PERFORM.

Avoiding GOTO With PERFORM

It is recommended that you do not include a GOTO command within the scope of a PERFORM command. See [GOTO](#) on page 101 for information on the incompatibility of the PERFORM and GOTO commands.

REPEAT

The REPEAT command enables you to loop through a block of code. REPEAT defines the beginning of the block, and ENDREPEAT defines the end. You control the loop by specifying the number of loop iterations, and/or the conditions under which the loop terminates. You can also define counters to control processing within the loop, for example incrementing a row counter to loop through the rows of a stack.

Syntax: How to Use the REPEAT Command

The syntax of the REPEAT command is:

```
REPEAT {int|ALL|WHILE condition|UNTIL condition} [counter [/fmt] =
init_expr] [;]
    command
    .
    .
    .
ENDREPEAT [counter [/fmt]=increment_expr;...]
```

where:

int

Specifies the number of times the REPEAT loop is to run. The value of *int* can be an integer constant, an integer field, or a more complex expression that resolves to an integer value. If you use an expression, the expression should resolve to an integer, although other types of expressions are possible. If the expression resolves to a floating-point or packed-decimal value, the decimal portion of the value will be truncated. If it resolves to a character representation of a numeric value, it will be converted to an integer value.

Expressions are described in [Expressions Reference](#) on page 21.

ALL

Specifies that the loop is to repeat indefinitely, terminating only when a GOTO EXITREPEAT command is issued from within the loop.

WHILE

Specifies that the WHILE condition is to be evaluated prior to each execution of the loop. If the condition is true, the loop is entered. If the condition is false, the loop terminates and control passes directly to the command immediately following ENDREPEAT. If the condition is false when the REPEAT command is first executed, the loop is never entered.

UNTIL

Specifies that the UNTIL condition is to be evaluated prior to each execution of the loop. If the condition is false, the loop is entered. If the condition is true, the loop terminates and control passes directly to the command immediately following ENDREPEAT. If the condition is true when the REPEAT command is first executed, the loop is never entered.

condition

Is a valid Maintain Data expression that can be evaluated as true or false (that is, a Boolean expression).

counter

Is a variable that you can use as a counter within the loop. You can assign the initial value of the counter in the REPEAT command, or in a COMPUTE command issued prior to the REPEAT command. You can increment the counter at the end of each loop iteration in the ENDREPEAT command. You can also change the value of the counter in a COMPUTE command within the loop. You can refer to the counter throughout the loop, including:

- ❑ Inside the loop, as a stack subscript.
- ❑ Inside the loop, in an expression.

- ❑ In a WHILE or UNTIL condition in the REPEAT command.

fmt

Is the format of the counter. It can be any valid format except for TX. The format is required only if you are defining the variable in this command.

init_expr

Is an expression whose value is assigned to the counter before the first iteration of the loop. It can be any valid Maintain Data expression.

increment_expr

Is an expression whose value is assigned to the counter at the end of each complete loop iteration. It can be any valid Maintain Data expression.

command

Is one or more Maintain Data commands, except for CASE, DECLARE, DESCRIBE, END, MAINTAIN, and MODULE.

;

Terminates the command. If you do not specify a counter, the semicolon is optional but recommended. Including it allows for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Simple Loops

The following code has a loop that executes ten or fewer times. The REPEAT line initiates the loop. The number 10 indicates that the loop will run ten times, barring any conditions or commands to exit the loop. The ON NONEXT GOTO EXITREPEAT command causes the loop to be exited when there are no more instances of Sales in the data source. The COMPUTE command calculates TotSales within an execution of the loop. The ENDREPEAT command is the boundary for the loop. Commands after ENDREPEAT are not part of the loop.

Because there is no loop counter, there is no way to know which repetition of the loop is currently executing:

```
COMPUTE TotSales = 0;
REPEAT 10;
  NEXT Sales;
  ON NONEXT GOTO EXITREPEAT;
  COMPUTE TotSales = TotSales + Sales;
ENDREPEAT
```

Example: Specifying Loop Iterations

You can control the number of times that the flow of control cycles through the loop by specifying the number of iterations. For example:

```
REPEAT 27;
```

You can also specify a condition that must be true or false for looping to continue:

```
REPEAT WHILE Rows GT 15;
```

Example: Repeating a Loop a Variable Number of Times

The REPEAT variable construct indicates that the loop is repeated the number of times indicated by the value of the variable. In this example, Stk1 is the name of a stack and FocCount is a stack variable that contains the number of rows in the stack. The loop executes a variable number of times based on the value of Stk1.FocCount:

```
FOR ALL NEXT Country INTO Stk1;  
COMPUTE Cnt/I4 = 1;  
REPEAT Stk1.FocCount;  
    TYPE "Country = <Stk1(Cnt).Country";  
    COMPUTE Cnt = Cnt + 1;  
ENDREPEAT
```

Example: REPEAT WHILE and UNTIL

The REPEAT WHILE construct indicates that the loop should be repeated as long as the expression is true. Once the expression is no longer true, the loop is exited. In this example, the loop will be executed Stk1.FocCount number of times. Stk1 is the name of a stack and FocCount is a stack variable that contains the number of rows in the stack:

```
FOR ALL NEXT Country INTO Stk1;  
COMPUTE CNT/I4 = 1;  
REPEAT WHILE Cnt LE Stk1.FocCount;  
    TYPE "Country = <Stk1(Cnt).Country ";  
    COMPUTE Cnt = Cnt + 1;  
ENDREPEAT
```

The REPEAT UNTIL construct indicates that the loop is repeated as long as the expression is not true. Once the expression is true, the loop is exited. In this example, the loop is executed Stk1.FocCount number of times. Stk1 is the name of a stack and FocCount is a stack variable that contains the number of rows in the stack. The COMPUTE increments the counter, although this could have been specified on the ENDREPEAT command. ENDREPEAT indicates the end of the loop:

```

FOR ALL NEXT Country INTO Stk1;
COMPUTE Cnt/I4 = 1;
REPEAT UNTIL Cnt GT Stk1.FocCount;
    TYPE "Country = <Stk1(Cnt).Country";
    COMPUTE Cnt = Cnt + 1;
ENDREPEAT

```

Example: Establishing Counters

You can use as many counters as you need in each loop. The only restriction is that all counter initializations performed in the REPEAT command must fit on the single line of the REPEAT command, and all counter incrementation performed in the ENDREPEAT command must fit on the single line of the ENDREPEAT command. You can avoid the single-line limitation by defining and incrementing counters using COMPUTE commands. It is legitimate, however, to have a REPEAT loop and never refer to any counter within the loop. If this is done, the same row of data is always worked on and unexpected results can occur.

The following examples do not have any index notation on the Stkemp stack, so each NEXT puts data into the same row of the stack. In other words, INTO Stkemp is the same as INTO Stkemp(1). Row one is always referenced because, by default, if there is a stack name without a row number, the default row number of one is used.

```

REPEAT 10;
    NEXT Emp_ID INTO Stkemp;
    .
    .
    .
ENDREPEAT

```

is the same as:

```

REPEAT 10 Cnt/I4=1;
    NEXT Emp_ID INTO Stkemp;
    .
    .
    .
ENDREPEAT Cnt=Cnt+1;

```

To resolve this problem, the REPEAT loop can establish counters and how they are incremented. Inside the loop, individual rows of a stack can be referenced using one of the REPEAT loop counters. The REPEAT command can be used to initialize many variables that will be used in the loop. For example

```

REPEAT 100 Cnt/I4=1; Flag=IF Factor GT 10 THEN 2 ELSE 1;

```

or:

```

REPEAT ALL Cnt = IF Factor GT 10 THEN 1 ELSE 10;

```

On the ENDREPEAT command, the counters are incremented by whatever calculations follow the keyword ENDREPEAT. Two examples are

```
ENDREPEAT Cnt = Cnt + 1; Flag = Flag*2;
```

and:

```
ENDREPEAT Cnt=IF Department EQ 'MIS' THEN Cnt+5 ELSE Cnt+1;
```

The following code sets up a repeat loop and computes the variable New_Sal for every row in the stack. The REPEAT line initiates the loop. The ALL indicates that the loop continues until a command in the loop tells the loop to exit. A GOTO EXITREPEAT command is needed in a loop when REPEAT ALL is used. The Cnt = 1 initializes the counter to 1 the first time through the loop. The COMPUTE command calculates a five percent raise. It uses the REPEAT counter (Cnt) to access each row in the stack one at a time. The counter is checked to see if it is greater than or equal to the number of rows in the Stkemp stack. The stack variable FocCount always contains the value of the number of rows in the stack. After every row is processed, the loop is exited.

The ENDREPEAT command contains the instructions for how to increment the counter:

```
REPEAT ALL Cnt/I4=1;
  COMPUTE Stkemp(Cnt).NewSal/D12.2=Stkemp(Cnt).Curr_Sal * 1.05;
  IF Cnt GE Stkemp.FocCount THEN GOTO EXITREPEAT;
ENDREPEAT Cnt=Cnt+1;
```

Example: Nested REPEAT Loops

REPEAT loops can be nested. This example shows one repeat loop nested within another loop. The first REPEAT command indicates that the loop will run as long as the value of A is less than 3. It also initializes the counter A to 1. The second REPEAT command indicates that the nested loop will run until the value of B is greater than 4. It initializes the counter B to 1. Two ENDREPEAT commands are needed, one for each REPEAT command. Each ENDREPEAT increments its respective counters.

```
REPEAT WHILE A LT 3; A/I4 = 1;
  TYPE "In A loop with A = <A";
  REPEAT UNTIL B GT 4; B/I4 = 1;
    TYPE "    ***In B loop with B = <B ";
  ENDREPEAT B = B + 1;
ENDREPEAT A = A + 1;
```

The output of these REPEAT loops would look like the following:


```

In A loop with A = 1
  ***In B loop with B = 1
  ***In B loop with B = 2
  ***In B loop with B = 3
  ***In B loop with B = 4
In A loop with A = 2
  ***In B loop with B = 1
  ***In B loop with B = 2
  ***In B loop with B = 3
  ***In B loop with B = 4

```

Reference: **Usage Notes for REPEAT**

The actual number of loop iterations can be affected by other phrases and commands in the loop. The loop can end before completing the specified number of iterations if it is terminated by a WHERE or UNTIL condition, or by a GOTO EXITREPEAT command issued within the loop.

Reference: **Commands Related to REPEAT**

- ❑ **COMPUTE.** Is used to define user-defined variables and assign values to existing variables.
- ❑ **GOTO.** Transfers control to another function or to the end of the current function.

Branching Within a Loop

There are two branching instructions that facilitate the usage of REPEAT and ENDREPEAT to control loop iterations:

- ❑ **GOTO ENDREPEAT.** Causes a branch to the end of the repeat loop and executes any computes on the ENDREPEAT line.
- ❑ **GOTO EXITREPEAT.** Causes the loop to be exited and goes to the next logical instruction after the ENDREPEAT.

Example: **Terminating the Loop From the Inside**

You can terminate a REPEAT loop by branching from within the loop to outside the loop. When you issue the command GOTO EXITREPEAT, Maintain Data branches to the command immediately following the ENDREPEAT command. It does not increment counters specified in the ENDREPEAT command. For example:

```
REPEAT ALL;  
.  
.  
.  
    GOTO EXITREPEAT;  
.  
.  
.  
ENDREPEAT
```

REPOSITION

For a specified segment and each of its descendants, the REPOSITION command resets the current position to the beginning of chain for that segment. That is, each segment is reset to just prior to the first instance.

Most data source commands change the current segment position to the instance that they most recently accessed. To search an entire data source or path for records, start at the beginning of the data source or path by first issuing the REPOSITION command.

Syntax: How to Use the REPOSITION Command

The syntax of the REPOSITION command is

```
REPOSITION segment_spec [ ; ]
```

where:

segment_spec

Is the name of a segment or the name of a field in a segment. The specified segment and all of its descendants are repositioned to the beginning of the segment chain.

;

Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Using REPOSITION

The following example repositions the root segment and all of the descendant segments of the Employee data source:

```
REPOSITION Emp_ID;
```

The next example repositions both the SallInfo and Deduct segments in the Employee data source:

```
REPOSITION Pay_Date;
```

Reference: Commands Related to REPOSITION

- ❑ **NEXT.** Starts at the current position and moves forward through the data source and can retrieve data from one or more records.
- ❑ **MATCH.** Searches entire segments for a matching field value and can retrieve an exact match in the data source.

REVISE

The REVISE command reads a stack of transaction data and writes it to a data source, inserting new segment instances and updating existing instances.

REVISE combines the functionality of the INCLUDE and UPDATE commands. It reads each stack row and processes each segment in the specified path using the following logic:

```
MATCH key
ON MATCH UPDATE fields
ON NOMATCH INCLUDE segment
```

You specify a path running from an anchor segment to a target segment. For each segment in the path, REVISE matches the instance of the segment in the stack against the corresponding instances in the data source. If the keys of an instance fail to find a match in the data source, REVISE adds the instance. If an instance does find a match, REVISE updates it using the fields that you have specified. The values that REVISE writes to the data source are provided by the stack.

Data source commands treat a unique segment as an extension of its parent, so that the unique fields seem to reside in the parent. Therefore, when REVISE adds an instance to a segment that has a unique child, it automatically also adds an instance of the child.

If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor, or provide ancestor segment key values in the source stack. This enables REVISE to navigate from the root to the first instance of the anchor segment.

Syntax: How to Use the REVISE Command

The syntax of the REVISE command is

```
[FOR {int|ALL} REVISE data_spec [FROM stack [(row)]] [;]
```

where:

FOR

Indicates that an integer or ALL will be used to specify how many stack rows to write to the data source.

If you specify FOR, you must also specify a source stack using the FROM phrase. If you omit FOR, REVISE defaults to writing one row.

int

Is an integer expression that specifies the number of stack rows to write to the data source.

ALL

Specifies that all of the rows of the stack are to be written to the data source.

data_spec

Identifies the path to be written to the data source and the fields to be updated:

1. Specify each field that you want to update in existing segment instances. You can update only non-key fields. Because a key uniquely identifies an instance, keys can be added and deleted but not changed.
2. Specify the path by identifying its anchor and target segments. You can specify a segment by providing its name or the name of one of its non-key fields.

If you have already identified the anchor and target segments in the process of specifying update fields, you do not need to do anything further to specify the path. Otherwise, if either the anchor or the target segment has not been identified using update fields, specify it using its segment name.

FROM

Indicates that the transaction data will be supplied by a stack. If this is omitted, the transaction data is supplied by the Current Area.

stack

Is the name of the stack whose data is being written to the data source.

row

Is a subscript that specifies the first stack row to be written to the data source. If omitted, it defaults to 1.

;

Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Using REVISE

In the following example the user is able to enter information for a new employee, or change the last name of an existing employee. Existing employee records are displayed in a grid. All of the information is stored in a stack named EmpStk.

```
MAINTAIN FILE EMPLOYEE
FOR ALL NEXT Emp_ID INTO EmpStk;
Winform Show GetData;

CASE Alter_Data
FOR ALL REVISE Last_Name FROM EmpStk;
ENDCASE

END
```

When the function Alter_Data is called from an event handler of a form, the REVISE command reads EmpStk and tries to find the Emp_ID of each row in the Employee data source. If Emp_ID exists in the data source, REVISE updates the Last_Name field of that segment instance. If it does not exist, then REVISE inserts a new EmplInfo instance into the data source, and writes the fields of EmplInfo from the stack to the new instance.

Reference: Usage Notes for REVISE

Maintain Data requires that the data sources to which it writes have unique keys.

Reference: Commands Related to REVISE

- INCLUDE.** Adds new segment instances to a data source.
- UPDATE.** Updates data source fields.
- COMMIT.** Makes all data source changes since the last COMMIT permanent.
- ROLLBACK.** Cancels all data source changes made since the last COMMIT.

ROLLBACK

The ROLLBACK command processes a logical transaction. A logical transaction is a group of data source changes that are treated as one. The ROLLBACK command cancels prior UPDATE, INCLUDE, and DELETE operations that have not yet been committed to the data source using the COMMIT command.

Syntax: How to Use the ROLLBACK Command

The syntax of the ROLLBACK command is

```
ROLLBACK [ ; ]
```

where:

```
;
```

Terminates the command. Although the semicolon is optional, you should include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Using ROLLBACK

This example shows part of a procedure where an employee ID needs to be changed. Because Emp_ID is a key, it cannot be changed. To accomplish this, it is necessary to collect the existing field values, make the necessary changes, delete the employee from the data source, and add a new segment instance.

The following shows partial code where the existing instance is deleted and a new one is added. If for some reason the INCLUDE does not work, the DELETE should not occur.

```
CASE Chngempid
DELETE Emp_ID;
IF FocError NE 0 PERFORM DeleteError;
INCLUDE Emp_ID Bank_Name Dat_Inc Type Pay_Date Ded_Code;
IF FocError NE 0 PERFORM Undo;
ENDCASE
```

```
CASE Undo
ROLLBACK;
ENDCASE
```

Reference: Usage Notes for ROLLBACK

- ❑ A ROLLBACK is automatically issued when a program is exited abnormally.
- ❑ A successful ROLLBACK issued in a called procedure frees the data source position maintained by that procedure and by all calling procedures.

- ❑ A ROLLBACK is automatically issued if an attempt to COMMIT fails.

DBMS Combinations

When an application accesses more than one DBMS (for example, DB2 and Teradata), ROLLBACK is treated as a broadcast rollback. There is no coordination between the different types of data sources, therefore the ROLLBACK might succeed against one type of data source but fail against another.

SAY

The SAY command writes messages to a file or to the default output device. You can use SAY for application debugging, such as tracing application control flow, and for recording an accounting trail. To display messages to application users, you should use forms, which provide superior display capabilities and better control than the SAY command.

Syntax: How to Use the SAY Command

The syntax of the SAY command is

```
SAY [TO ddname] expression [expression ...] ;
```

where:

TO ddname

Specifies the logical name of the file to which the SAY message is written. *ddname* is a character expression. If you supply a literal for *ddname*, it must be enclosed in single quotation marks (') or double quotation marks (").

You must define the logical name using a DYNAM command on z/OS before the SAY command is executed. In order to append to an existing file (for example, to write to a file from more than one procedure), specify the appropriate option in the DYNAM command.

If *TO ddname* is omitted, the message is written to the default output device of the environment in which the SAY command is issued.

In addition, if *TO ddname* is omitted and this procedure was called remotely (that is, called using a CALL *procname* command), the message will also be copied to the FocMsg stack of the calling procedure.

expression

Is any Maintain Data expression. Multiple expressions must be separated by spaces.

Each message is written on the current line, beginning in the column that follows the end of the previous message. When a message reaches the end of the current line in the file or display device, or encounters a line feed (the string `\n`) in the message text, the message stream continues in column 1 of the next line.

If you write to output devices that buffer messages before displaying them, you may need to end each message with an explicit line feed to force the buffer to display the last line of the message.

Note: Literals must be enclosed in single quotation marks (') or double quotation marks ("), while variables do not appear in quotation marks.

Reference: Commands Related to SAY

TYPE. Writes messages to a file or to a form.

Writing Segment and Stack Values

You can use the `SEG` and `STACK` prefixes to write the values of all the fields of a segment or columns of a stack to a message. This can be helpful when you write messages to log and checkpoint files.

`SEG.fieldname` inserts Current Area values for all of the fields of the specified segment.

`STACK.stackname(row)` inserts, for the specified stack, the values of the specified row.

Choosing Between the SAY and TYPE Commands

The rules for specifying messages using the `SAY` command are simpler and more powerful than those for the `TYPE` command. For example, you can include all kinds of expressions in a `SAY` command, but you can only include character string constants and scalar variables in a `TYPE` command.

Note that, unlike the `TYPE` command, the `SAY` command does not generate a default line feed at the end of each line.

SET

You can change parameters that control output, work areas, and other features in your Reporting Server environment by using the `SET` command from an App Studio procedure (not a Maintain Data procedure).

This command is outside the Maintain Data language, but is described here for your convenience, since many of these settings affect how Maintain Data behaves.

However, you can change a limited number of `SET` parameters from within a Maintain Data procedure using `SYSMGR.FOCSET`. For more information, see [SYS_MGR.FOCSET](#) on page 159.

For a list of SET parameters, see *Customizing Your Environment* in the *Developing Reporting Applications* manual.

Syntax: **How to Use SET Parameters**

The syntax is

```
SET parameter = option[, parameter = option,...]
```

where:

parameter

Is the setting to change.

option

Is one of a number of options available for each parameter.

You can set several parameters in one command by separating each with a comma.

You may include as many parameters as you can fit on one line. Repeat the SET keyword for each new line.

Note: This syntax is valid *only* in an App Studio procedure.

Syntax: **How to Use SET Parameters in a Request**

Many SET commands that change system defaults can be issued from within TABLE requests. SET used in this manner is temporary, affecting only the current request.

The syntax is

```
ON TABLE SET parametervalue [AND parametervalue ...]
```

where:

parameter

Is the system default to change.

value

Is an acceptable value that will replace the default value.

Note: This syntax is valid *only* in an App Studio report procedure.

showLayer

Note: The SetLayer command in legacy Maintain has been replaced by the IbComposer_showLayer command in Maintain Data. The IbComposer_showLayer command allows layers to be set as visible or invisible at run time. It is used in a JavaScript event.

Syntax: **How to Use the showLayer Command**

```
IbComposer_showLayer(layername, bShow);
```

where:

layername

Alphanumeric

Is the name of the layer, which will be shown or hidden.

bShow

Is an operator that can be set to true to show the layer or false to hide it.

Example: **Showing or Hiding a Layer**

```
function button1_onclick(event) {  
    var eventObject = event ? event : window.event;  
    var ctrl = eventObject.target ? eventObject.target :  
    eventObject.srcElement;  
    // TODO: Add your event handler code here  
    IbComposer_showLayer('Customers', 'true');  
}
```

STACK CLEAR

STACK CLEAR clears the contents of each of the stacks listed, so that each stack has no rows. This sets the FocCount variable to zero (0) and FocIndex variable to one (1) for each stack.

Syntax: **How to Use the STACK CLEAR Command**

The syntax of the STACK CLEAR command is

```
STACK CLEAR stacklist [;]
```

where:

stacklist

Specifies the stacks to be initialized. Stack names are separated by blanks.

;

Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Using STACK CLEAR

The following initializes the Emp stack:

```
STACK CLEAR Emp;
```

The next example initializes both the Emp and the Dept stacks:

```
STACK CLEAR Emp Dept;
```

STACK SORT

The STACK SORT command enables you to sort the contents of a stack in ascending or descending order based on the values in one or more columns.

Syntax: How to Use the STACK SORT Command

The syntax for the STACK SORT command is

```
STACK SORT stackname BY [HIGHEST] column [BY [HIGHEST] column ...] [;]
```

where:

stackname

Specifies the stack to be sorted. The stack name cannot be subscripted with a row range in order to sort only part of the stack.

HIGHEST

If specified, sorts the stack in descending order. If not specified, the stack is sorted in ascending order.

column

Is a stack column name or a computed field value. At least one column name must be specified. The column must exist in the specified stack.

;

Terminates the command. Although the semicolon is optional, it is recommended that you include it to allow for flexible syntax and better processing. For more information about the benefits of including the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Using STACK SORT

The following sorts the Emp stack using the values in the stack column Emp_ID:

```
STACK SORT Emp BY Emp_ID;
```

The following sorts the Emp stack so that the employees with the highest Salary are placed first in the stack:

```
STACK SORT Emp BY HIGHEST Salary;
```

The next example sorts the stack by Department. Within Department, the rows are ordered by highest Salary:

```
STACK SORT Emp BY Department BY HIGHEST Salary;
```

Sorting Data With the Using CASE_INSENSITIVE Parameter

You can sort data in a stack without considering case by using the CASE_INSENSITIVE parameter.

Syntax: How to Sort Data in a Stack With the Using CASE_INSENSITIVE Parameter

To sort data in a stack and ignore case-sensitivity:

1. Add the *using CASE_INSENSITIVE* tablename parameter after the BY fields:

```
STACK SORT stackname BY [HIGHEST] column [BY [HIGHEST] column ...]
[using CASE_INSENSITIVE] [;]
```

2. Import the MNTUWS function library.
3. Add the following command to the Top case of the Maintain Data procedure:

```
perform prep_CASE_INSENSITIVE
```

For example:

```
MAINTAIN
MODULE IMPORT (mntuws)
$$Declarations
CASE Top
PERFORM prep_CASE_INSENSITIVE;
COMPUTE STK(1).NAME/A10="GEORGE";
STK(2).NAME="Bernard";
STK(3).NAME="Shaw";
.
.
.
STACK SORT STK BY NAME using CASE_INSENSITIVE;
.
.
.
```

Note: For a back-end procedure without a Top case, the `perform prep_CASE_INSENSITIVE` command is not required.

When using `MNTCON MATCH_CASE ON`, the case of the code must be the same as it appears above.

SYS_MGR

The `SYS_MGR` global object provides functions and variables that control the environment for your Maintain application. It can help developers ensure the most efficient interaction between Maintain and DBMS servers, as well as manage App Studio environment and run-time variables.

For use with relational data sources, `SYS_MGR` can be used from within Maintain procedures to:

- ❑ Deactivate preliminary database operation checking by Maintain before an update and rely on the DBMS to perform its own internal checking, thus reducing processing time and resources (`SYS_MGR.PRE_MATCH`).
- ❑ Retrieve DBMS error codes, allowing developers to code applications to efficiently recover from error conditions (`SYS_MGR.DBMS_ERRORCODE`).
- ❑ Issue native SQL commands directly from a Maintain procedure (`SYS_MGR.ENGINE`).

The `SYS_MGR` syntax can also be used from within a Maintain procedure to:

- ❑ Issue system settings for Reporting Servers dynamically (`SYS_MGR.FOCSET`).
- ❑ Set values for Userid and Password for target servers before issuing an `EXEC AT` or `CALL AT`.
- ❑ Retrieve input parameter values passed at invocation time (`SYS_MGR.GET_INPUTPARAMS_COUNT`, `SYS_MGR.GET_NAMEPARM`, `SYS_MGR.GET_POSITIONPARM`).

Note: All `sys_mgr` function and variable names are case-insensitive. You can use `SYS_MGR.DBMS_ERRORCODE`, `sys_mgr.dbms_errorcode`, `sys_mgr.DBMS_ErrorCode`, and so on, interchangeably.

SYS_MGR.DBMS_ERRORCODE

The `SYS_MGR.DBMS_ERRORCODE` command enables you to retrieve error codes returned by the DBMS server and then take appropriate action. For example, a developer might want to take a different course of action for an `INSERT` that fails because the user does not have `INSERT` rights versus a referential integrity failure.

Note:

- ❑ The return codes are DBMS specific. The DB2 return codes do not match the Oracle return codes. Moreover, DBMS vendors have been known to change the return codes on release boundaries. You should clearly document that you are using this feature so sufficient testing can be done before rolling in a new DBMS.
- ❑ DBMS_ERRORCODE is local to the current Maintain procedure.

Syntax: **How to Use SYS_MGR.DBMS_ERRORCODE**

The syntax is

```
SYS_MGR.DBMS_ERRORCODE ;
```

Example: **Retrieving an Error Code From a DBMS**

For example, the following Maintain code will retrieve an error code from a DBMS, and if it is a specific code, branches to some appropriate code:

```
Compute ErrCode/a3 = SYS_MGR.DBMS_ERRORCODE ;  
If ErrCode EQ '515' goto BadInsert;
```

SYS_MGR.ENGINE

You can issue DBMS commands directly (SQL Passthru) from a Maintain procedure using the SYS_MGR.ENGINE command.

Note: Problems with direct commands are not reported in FOCERROR. You will need to use DBMS_ERRORCODE to determine the success or failure of these commands.

Syntax: **How to Use the SYS_MGR.ENGINE Command**

The syntax for the SYS_MGR.ENGINE command is

```
SYS_MGR.ENGINE("enginename", "command");
```

where:

enginename

Is the name of the RDBMS to which you are passing the command. For a complete list of the possible values, see the *Adapter Administration* manual.

command

Is any valid SQL command, including CREATE, DROP, and INSERT.

Example: Issuing the DROP TABLE Command

The following command drops the table NYACCTS. The error code is saved in a variable named rc.

```
Compute rc/i8;
rc = sys_mgr.engine("SQLMSS", "DROP TABLE NYACCTS");
Type "Return Code=<<rc DBMS Err=<<SYS_MGR.DBMS_ERRORCODE" ;
```

Example: Setting Connection Attributes for an MS SQL Server

```
Compute rc/i8;
rc=sys_mgr.engine("SQLMSS","set connection_attributes mssxyz/ibiusr1,foo"
);
Type "RC from set is <<rc DBMS Err=<<SYS_MGR.DBMS_ERRORCODE";
```

Example: Inserting a Row Into a Table (MS SQL)

```
Compute rc/i8;
Type "Inserting row into table MNTTAB2 ";
rc=sys_mgr.engine("SQLMSS","insert into mntbtab2
values('X2','XDAT2222');");
Type"ReturnCode=<<rc DBMS Err=<<SYS_MGR.DBMS_ERRORCODE";
```

You will need to test the return code to determine whether the record was inserted successfully (RC = 0).

If you are using MS SQL, and the value you wanted to insert was a duplicate record, you would expect to see the following return codes:

```
Return Code= -1 DBMS Err= 2627
```

SYS_MGR.FOCSET

Using SYS_MGR.FOCSET, you can set certain environment settings for the Reporting Server. Issue this command from a Maintain procedure to set the desired environment variable, then use a local call from the same procedure to use that setting for your Maintain operations. See *Customizing Your Environment* in the *Developing Reporting Applications* manual for a complete description of the environment settings.

Syntax: **How to Use the SYS_MGR.FOCSET Command**

The syntax is

```
SYS_MGR.FOCSET("parm", "value")
```

where:

parm

Is one of the following supported SET commands:

```
CDN  
COMMIT  
DATEDISPLAY  
DEFCENT (DFC)  
EMGSRV  
LANGUAGE  
MESSAGE  
NODATA  
TRACEON  
TRACEOFF  
TRACEUSER  
WARNING  
YRTHRESH  
PASS  
USER
```

In addition, the parameter *maintain_warning* is included in this command set in order to allow Maintain warning messages to be suppressed.

value

Is an appropriate setting for that command.

Example: **Setting DEFCENT From a Maintain Procedure**

The following code

```
MAINTAIN  
COMPUTE MYDATE/YMD;  
SYS_MGR.FOCSET("DEFCENT", "21");  
COMPUTE DATE1/YMD='90/01/01';  
COMPUTE MYDATE=DATE1;  
TYPE "After setting DEFCENT=21, MYDATE=<<MYDATE";  
END
```

produces the following output:

```
After setting DEFCENT=21, MYDATE=2190/01/01
```


Example: Setting PASS From a Maintain Procedure

The following code will set the password to DBAUSER2:

```
SYS_MGR.FOCSET('PASS', 'DBAUSER2.');
```

Note: When setting a password for DBA access, keep in mind that the last value set from within the application will be in effect for all transactions for that end user session.

Example: Setting maintain_warning From a Maintain Procedure

The following code allows you to display your own error message:

```
CASE TEST1
COMPUTE DATE1/A10;
COMPUTE DATE2/MDYY
SYS_MGR.FOCSET("MAINTAIN_WARNING ", "OFF")
COMPUTE DATE2 = DATE1;
IF DATE2 = '' THEN
COMPUTE MSG = 'Date is not valid';
ENDCASE
```

To set the Maintain warning messages to on, issue the command:

```
SYS_MGR.FOCSET("MAINTAIN_WARNING ", "ON")
```

SYS_MGR.GET_INPUTPARAMS_COUNT

Used in conjunction with the MNTCON EX or MNTCON RUN -v syntax, the SYS_MGR.GET_INPUTPARAMS_COUNT function retrieves the number of positional parameters passed when invoking a Maintain procedure. If the function is not successful, FOCERROR is set to -1. For information on retrieving the value of a positional parameter, see the SYS_MGR.GET_POSITIONPARM function. See the MNTCON EX and MNTCON RUN commands for information on using the -v option to pass parameters.

Syntax: How to Use the SYS_MGR.INPUTPARAMS_COUNT Command

The syntax for the SYS_MGR.GET_INPUTPARAMS_COUNT command is

```
Var/In = SYS_MGR.GET_INPUTPARAMS_COUNT();
```

where:

Var/In

Is the name of the variable with an integer format that you are assigning to the output of the function.

Example: Retrieving the Number of Positional Parameters Passed to a Maintain Procedure

```
MNTCON EX START1 -v abc, '24 Houston Center'
```

Target Maintain procedure START1 could include:

```
Posvar/i2=sys_mgr.Get_InputParams_count();
```

Here it returns 2 for the computed field Posvar, for positional parameters abc and 24 Houston Center.

SYS_MGR.GET_NAMEPARAM

Used in conjunction with the MNTCON EX or MNTCON RUN -v syntax, the SYS_MGR.GET_NAMEPARAM function returns the value of a keyword parameter passed at the time the Maintain procedure was invoked. If the function is not successful, FOCERROR is set to -1. See also the MNTCON EX and MNTCON RUN commands using the -v option to pass parameters.

Syntax: How to Use the SYS_MGR.GET_NAMEPARAM Command

The syntax for the SYS_MGR.GET_NAMEPARAM command is

```
MyParm/ format = SYS_MGR.GET_NAMEPARAM( 'ParmName' )
```

where:

MyParm/ format

Is the name of the variable or format that you are assigning to the output of the function.

ParmName

Is the actual keyword parameter name used when passing the value.

Note: The SYS_MGR.GETNAME_PARM function is case-sensitive. Use the same case for the parameter name when retrieving the value as used when passing it.

Example: Retrieving the Value for a Keyword Parameter Passed to a Maintain Procedure

```
MNTCON EX START2 -v ADDR='Cape Canaveral', COUNTRY=USA
```

Target Maintain procedure START2 could include:

```
Address/a0=sys_mgr.Get_NameParm( 'ADDR' );
```

Here it returns Cape Canaveral to the variable Address.

SYS_MGR.GET_POSITIONPARM

Used in conjunction with the MNTCON EX or MNTCON RUN `-v` syntax, the SYS_MGR.GET_POSITIONPARM function retrieves the value of positional parameters passed when invoking a Maintain procedure. If the function is not successful, FOCERROR is set to -1. See also the SYS_MGR.GET_INPUTPARAMS_COUNT function to retrieve the number of a positional parameter, and the MNTCON EX and MNTCON RUN commands using the `-v` option to pass parameters.

Syntax: How to Use the SYS_MGR.GET_POSITIONPARM Command

The syntax for the SYS_MGR.GET_POSITIONPARM command is

```
Var/An = SYS_MGR.GET_POSITIONPARM(i);
```

where:

Var/An

Is the name of the variable, declared with an alphanumeric format (for example, A0) that you are assigning to the output of the function.

i

Is the position number of the variable to retrieve.

Example: Retrieving the Value of a Positional Parameter Passed to a Maintain Procedure

```
MNTCON EX START3 -v abc, '24 Houston Center'
```

Target Maintain procedure START3 could include:

```
Parm1/a0=sys_mgr.Get_PositionParm(2);
```

Here it returns 24 Houston Center, the value of the second keyword parameter, for computed field Parm1.

SYS_MGR.PRE_MATCH

By default, Maintain first ensures a database row exists before it updates or deletes it and ensures a database row does NOT exist before including a new row. For example, when Maintain processes an INCLUDE, it first issues:

```
SQL SELECT keyfld FROM tablename WHERE keyfld = keyvalue;
```

Then, it only proceeds with the SQL INSERT if the SELECT returned no rows. Many applications are structured so that the designer knows that the row does not exist, so the preliminary SELECT is not needed.

The same is true for DELETE and UPDATE. Only the SELECT must return a row before MAINTAIN continues with the SQL DELETE or SQL UPDATE.

When the application warrants it, you can turn off the preliminary SELECT against relational databases by changing the value of SYS_MGR.PRE_MATCH. For high volume transactions, this can positively affect performance.

Note:

- ❑ Since you are not checking to see if the row exists or not, it is important to write code that catches errors by inspecting FOCERROR.
- ❑ The PRE_MATCH setting is local to the current MAINTAIN procedure. Changing it does not change the value in the parent procedure or in subsequently called procedures.

Syntax: **How to Set PRE_MATCH**

The syntax is

```
SYS_MGR.SET_PRE_MATCH{0|1};
```

or

```
SYS_MGR.PRE_MATCH = {0|1}
```

where:

0

Disables prematching.

1

Turns on prematching.

To check the current setting for pre-match, use:

```
SYS_MGR.GET_PRE_MATCH();
```

or

```
SYS_MGR.PRE_MATCH;
```

Example: **Setting PRE_MATCH Off**

Suppose you have a Maintain procedure with the following code:

```
SYS_MGR.PRE_MATCH = 0;  -* stop pre-selecting
FOR ALL INCLUDE PRODUCTS FROM PRODSTACK;
SYS_MGR.PRE_MATCH = 1;  -* restore
```

If PRODSTACK has 5000 rows, setting PRE_MATCH to 0 before the INCLUDE reduces the number of database engine interactions from 10,000 to 5,000.

TYPE

The TYPE command writes messages to a file, a web browser, or the Maintain Data output window. The TYPE command is helpful for application debugging, such as tracing application flow-of-control, and for recording an accounting trail. To display messages to application users, it is recommended that you use forms, which provide superior display capabilities and better control than the TYPE command.

Syntax: How to Use the TYPE Command

The syntax of the TYPE command is

```
TYPE [ON ddname] "message" [[|] "message"] ... [i]
```

where:

ON ddname

Specifies the logical name of the file that the TYPE message is written to when ON is specified. You must define the ddname (using a DYNAM or FILEDEF command) prior to the first usage. The message string can be up to 256 characters in length. The output starts in column 1. In order to append to an existing file or to write to a file from more than one procedure, append to the file by specifying the appropriate option in the DYNAM command.

In addition, if ON *ddname* is omitted and this procedure was called remotely (that is, called using a CALL *procname* command), the message will also be copied to the FocMsg stack of the calling procedure.

message

Is the information to be displayed or written. The message must be enclosed in double quotation marks ("). The message can contain:

- Any literal text
- Variables
- Horizontal spacing information
- Vertical spacing information

The layout of the message is exactly what is specified.

;

Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the benefits of including the semicolon, see *Terminating Command Syntax* on page 17.

Reference: Commands Related to TYPE

SAY. Writes messages to a file or to the server console. Messages can include multiple expressions of all types.

Including Variables in a Message

You can embed variables in a message by prefixing the variable with a left angle bracket (<). Unless the field name is the last item in the string, it must be followed by a space. Maintain Data does not include the angle bracket and space in the display. For example:

```
TYPE "Accepted: <Indata(Cnt).Fullname";
```

Embedding Horizontal Spacing Information

TYPE information can be placed in a specific column or can be moved a number of columns away from the current position. The following example

```
TYPE "<20 This starts 20 spaces over";
TYPE "Skip <+8 8 spaces within text";
TYPE "Back up <-4 4 spaces and overwrite";
```

results in:

```

                This starts 20 spaces over
Skip          8 spaces within text
Back4 spaces and overwrite
```

Embedding Vertical Spacing Information

Lines can be skipped by supplying a left angle bracket (<), slash (/), and the number of lines to be advanced. If the line advance specification is at the beginning of the line, the specified number of lines are advanced before the following text.

```
TYPE "</3 Displays 3 blank lines" |
" before this line";
```

If </number is encountered in the middle of the line, the line feed occurs when </number is encountered.

```
TYPE "This will </2 leave one" |
" blank line before the word leave";
```

Coding Multi-Line Message Strings

Sometimes, a message string needs to be coded on more than one line of a TYPE command. This can occur if indented TYPE lines, spacing information, or field prefixes extend the message string beyond the end of the line. You can wrap a message string onto the next line of a TYPE command if you:

1. End the first line with an ending double quotation mark ("), followed by a vertical bar (|).
2. Begin the second line with a double quotation mark ("). For example:

```
TYPE "Name: <Employee(Cnt).First_Name" |
"<Employee(Cnt).Last_Name" |
"Salary: <Employee(Cnt).Salary";
```

Justifying Variables and Truncating Spaces

To either truncate or display trailing spaces within a field, a left angle bracket (<) or a double left angle bracket (<<) may be used, respectively. For character fields, the field values are always left justified. For example:

```
TYPE "*** <Car.Country ***";
TYPE "*** <<Car.Country ***";
```

produces:

```
*** ENGLAND***
*** ENGLAND ***
```

For numeric fields, the left angle bracket causes the field values to be left justified, and trailing spaces are truncated. The double left angle bracket causes the field values to be right justified and leading spaces are displayed.

For example

```
TYPE "*** <Car.Seats ***"
TYPE "*** <<Car.Seats ***"
```

produces:

```
*** 4***
***   4***
```

Writing Information to a File

You can use TYPE commands to write information to a file. The following example writes every transaction record to a log file:

```
FOR ALL NEXT Emp_ID Last_Name First_Name INTO Stackemp;
COMPUTE Cnt=Cnt+1;
TYPE ON TransLog "<Stackemp(Cnt).Emp_ID " |
"<Stackemp(Cnt).Last_Name" |
"<Stackemp(Cnt).First_Name";
```

The next example places a message into an errors log file if the salary in the stack is greater than allowed:

```
IF Stackemp(Cnt).Curr_Sal GT Allowamt THEN TYPE ON ErrsFile
  "Salary for employee <Stackemp.Emp_ID" |
  "is greater than is allowed.";
```

The last example writes three lines to the file NoEmpl if the employee is not in the data source:

```
MATCH Emp_ID;
ON NOMATCH TYPE ON NoEmpl "<Emp_ID"
  "<Last_Name"
  "<First_Name";
```

UPDATE

The UPDATE command writes new values to data source fields using data from a stack or the Current Area. All of the fields must be in the same data source path. The key fields in the stack or Current Area identify which segment instances to update.

The segment containing the first update field is called the anchor. If the anchor segment is not the root, you must establish a current instance in each of the ancestor segments of the anchor, or provide ancestor segment key values in the source stack or Current Area. This enables UPDATE to navigate from the root to the first instance of the anchor segment.

Syntax: How to Use the UPDATE Command

The syntax of the UPDATE command is

```
[FOR {int|ALL}] UPDATE fields [FROM stack(row)] [;]
```


where:

FOR

Is used with *int* or ALL to specify how many rows of the stack to use to update the data source. When FOR is used, a FROM stack must be supplied. If no FOR prefix is used, the UPDATE works the same way that FOR 1 UPDATE works.

int

Is an integer constant or variable that indicates the number of rows to use to update the data source.

ALL

Specifies that the entire stack is used to update the corresponding records in the data source.

fields

Is used to specify every data source field to update. You cannot update key fields. All fields must be in the same path.

FROM

Is used to specify a stack containing records to insert. If no stack is specified, data from the Current Area is used.

stack

Is the name of the stack whose data is used to update the data source. Only one stack can be specified.

row

Is a subscript that specifies the first stack row to use to update the data source.

;

Terminates the command. Although the semicolon is optional, including it to allow for flexible syntax and better processing is recommended. For more information about the semicolon, see [Terminating Command Syntax](#) on page 17.

Example: Using UPDATE

The UPDATE command can be executed after a MATCH command finds a matching record. For example:

```
MATCH Emp_ID;
ON MATCH UPDATE Department Curr_Sal Curr_Jobcode Ed_Hrs FROM Chgemp;
```

Consider an application used when an employee changes his or her last name. The application user is prompted for the employee ID and new last name in a form. The user enters the name and triggers the ChngName function. If the employee is in the data source, ChngName updates the data source. If the employee is not in the data source, ChngName displays a message asking the user to try again.

```
CASE ChngName
REPOSITION Emp_ID;
MATCH Emp_ID;
ON MATCH BEGIN
    UPDATE Last_Name;
    COMMIT;
    Winform Close;
ENDBEGIN
ON NOMATCH BEGIN
    TYPE "Employee ID <Emp_ID was not found"
        "Try again";
ENDBEGIN
ENDCASE
```

The command can also be issued without a preceding MATCH. In this situation, the key field values are taken from the FROM stack or the Current Area and a MATCH is issued internally. When a set of rows is changed without first finding out if the set already exists in the data source, it is possible that some of the rows in the stack will be rejected. Upon the first rejection, the process stops and the rest of the set is rejected. For all rows to be accepted or rejected as a unit, the set should be treated as a logical unit of work, and a ROLLBACK issued if the entire set is not accepted.

Reference: Usage Notes for UPDATE

- Key fields cannot be updated.
- There can only be one input or FROM stack in an UPDATE command.
- When an UPDATE command is complete, the variable FocError is set. If the UPDATE is successful, FocError is set to zero (0). If the records do not exist, and are therefore unchanged, FocError is set to a non-zero value and (if the UPDATE is set-based) FocErrorRow is set to the number of the row that failed.
- Maintain Data requires that the data sources to which it writes have unique keys.

Reference: Commands Related to UPDATE

- COMMIT.** Makes all data source changes since the last COMMIT permanent.
- ROLLBACK.** Cancels all data source changes made since the last COMMIT.

Update and Transaction Variables

After the UPDATE is processed, the internal variable FocError is given a value. If the UPDATE is successful, FocError is zero (0). If the UPDATE fails (that is, the key values did not exist in the data source) FocError is set to a non-zero value, and (if the UPDATE was set-based) FocErrorRow is set to the number of the row that failed. If at COMMIT time there is a concurrency conflict, FocError and the internal variable FocCurrent are set to non-zero values.

Example: Using Stacks

In the following example, the user enters many employee IDs and new names at one time. Rather than performing a MATCH on each row in the stack, this function checks FocError after the UPDATE command. If FocError is zero (0), a COMMIT is issued and the function is exited. If FocError is non-zero, another function, which tries to clean up the data, is performed. The IF command, which starts at the beginning of the function, checks to see whether there are any rows in the stack. If the stack does contain have any rows, a form displays allowing the user to enter new data. If the stack contains rows, the user has made a mistake, so a different form displays allowing the user to edit the entered data.

The Maintain Data procedure contains:

```
STACK CLEAR Namechng;
PERFORM Chngname;
CASE Chngname
IF Namechng.FocCount LE 0
    THEN Winform Show Myform1;
    ELSE Winform Show Myform2;
FOR ALL UPDATE Last_Name FROM Namechng;
IF FocError EQ 0 BEGIN
    COMMIT;
    GOTO ENDCASE;
ENDBEGIN
PERFORM Fixup;
GOTO Chngname;
ENDCASE
```

Data Source Position

A Maintain Data procedure always has a position either in a segment or before the beginning of the chain. If positioned within a segment, the position is the last record successfully retrieved on that segment. If a retrieval operation fails, then the position of the data source remains unchanged.

If an UPDATE is successful, the data source position is changed to the last record it updated. If an UPDATE fails, the position is at the end of the chain because the MATCH prior to the UPDATE also fails.

Unique Segments

The UPDATE command treats fields in unique segments the same as fields in other types of segments.

Winform

The Winform command controls the forms that appear on the screen. Forms are used to edit and display data. They act as a user interface, whereas a procedure controls the application logic and use of data.

Syntax: How to Use the Winform Command

The syntax of the Winform command for displaying and controlling forms is

```
Winform command formname [i]
```

where *command* is one of the following:

Show

Makes the specified form active. It displays the form and transfers control to it, enabling an application user to manipulate the controls (of the form), such as buttons and fields.

Show_Active

Can be used for clarity. It is functionally identical to Show.

Show_Inactive

You can use this to change the initial properties of a form, and its controls, dynamically at run time before the form displays.

Reset

Resets a form and its controls to their original properties. All selectable controls, such as list boxes, check boxes, and radio buttons, return to their default selections.

Refresh

Repopulates the data values of the form as if control had returned to the form from an event handler, but without making the form active.

Close_All

Closes all forms. The form environment remains active.

Close

Closes the chain of forms from the currently active form back up to the specified form. If you do not specify a form, the command closes only the currently active form.

The close operation does the following:

- ❑ Passes control directly to the beginning of the chain, to the point just following the Winform Show command that called the specified form.
- ❑ Removes closed forms from the screen.

`Show_And_Exit`

Displays the specified form and then immediately terminates the application. This enables you to end an application while displaying a final form that remains on the screen. Any client-level logic, such as hypertext links and JavaScript functions, will remain active, but all native Maintain Data logic, such as event handlers, will not respond because the application has terminated.

formname

Is the name of the Maintain Data form.

Reference: Commands Related to Winform

- ❑ **NEXT.** Retrieves sets of data from a data source into a stack. You can then display the data in a form.
- ❑ **TYPE.** Displays messages on the screen or writes them to a file.

Displaying Default Values in a Form

If a form displays a variable that has not been assigned a value, the form will display the default value. The default value of the variable is determined by its data type and whether it was defined with the MISSING attribute:

Data Type	Default value without the MISSING attribute	Default value with the MISSING attribute
Character/Alphanumeric	space	null
Numeric	zero (0)	null
Date and time	space	null

A null value displays as a period (.) by default. You can specify a different character using the SET NODATA command.

WINFORM SET

The WINFORM SET command has been replaced by the COMPUTE command. For more information on the COMPUTE command, see [Using COMPUTE to Dynamically Change the Property of an Object](#) on page 72.

Ensuring Transaction Integrity

You are familiar with individual data source operations that insert, update, or delete data source segment instances. However, most applications are concerned with *real-world* transactions, like transferring funds or fulfilling a sales order, that each require several data source operations. These data source operations may access several data sources, and may be issued from several procedures. We call such a collection of data source operations a logical transaction (it is also known as a logical unit of work).

In this appendix:

- [Transaction Integrity Overview](#)
 - [Why Is Transaction Integrity Important?](#)
 - [Defining a Transaction](#)
 - [Evaluating Whether a Transaction Was Successful](#)
 - [Concurrent Transaction Processing](#)
 - [Ensuring Transaction Integrity for FOCUS Data Sources](#)
 - [Ensuring Transaction Integrity for DB2 Data Sources](#)
-

Transaction Integrity Overview

This topic describes how App Studio Maintain Data ensures transaction integrity at the application level. At the data source level, each database management system (DBMS) implements transaction integrity in its own way. For more information, see your DBMS vendor documentation for DBMS-specific information. For FOCUS data sources, this DBMS-specific information is presented in [Ensuring Transaction Integrity for FOCUS Data Sources](#) on page 184. For DB2, you can find some suggested strategies for writing Maintain Data transactions to DB2 data sources in [Ensuring Transaction Integrity for DB2 Data Sources](#) on page 191. For many other types of data sources, you can also apply the strategies described in [Ensuring Transaction Integrity for DB2 Data Sources](#) on page 191, changing DBMS-specific details when necessary.

Example: Describing a Transfer of Funds as a Logical Transaction

A banking application would define a transfer of funds from one account to another as one logical transaction comprising two update operations:

- Subtracting the funds from the source account (UPDATE Savings FROM SourceAccts).
- Adding the funds to the target account (UPDATE Checking FROM TargetAccts).

Procedure: How to Process a Logical Transaction

To process a logical transaction, follow these steps:

1. **DBMS requirements.** The database management system of your data sources (DBMS) may require that you perform some tasks to enable transaction integrity. For more information, see your DBMS vendor documentation for information.

You can set some native DBMS parameters using the SYS_MGR.FOCSET command. For more information, see [SYS_MGR.FOCSET](#) on page 159. You can also set some native DBMS parameters through FOCUS. See your server documentation.

For FOCUS data sources, you must set the COMMIT server parameter to ON, and issue a USE command to specify which FOCUS Database Server will manage concurrent access to the data source. For more information, see [Ensuring Transaction Integrity for FOCUS Data Sources](#) on page 184.

2. **Develop the transaction logic.** Code the data source commands and related logic that read from the data sources, write to the data sources, and evaluate the success of each data source command.
3. **Define the transaction boundary.** Code a COMMIT command, and any other supporting commands, to define the transaction boundary. For more information, see [Defining a Transaction](#) on page 177.
4. **Evaluate the success of the transaction.** Test the FocCurrent transaction variable to determine if the transaction was successfully written to the data source, and then branch accordingly. For more information, see [Evaluating Whether a Transaction Was Successful](#) on page 182.

Why Is Transaction Integrity Important?

The advantage of describing a group of related data source commands as one logical transaction is that the transaction is valid and written to the data source only if all of its component commands are successful. When you attempt to commit a transaction, you are ensured that if part of the transaction fails, none of the transaction will be written to the data source. This is called transaction integrity.

When is transaction integrity important? Whenever a group of commands are related and are only meaningful within the context of the group. In other words, whenever the failure of any one command in the transaction at commit-time would invalidate the entire transaction.

Transaction integrity is an uncompromising proposition: either all of the transaction is written to the data source when you commit it, or all of it is rolled back.

Example: **Why Transaction Integrity Is Essential to a Bank**

Consider a banking application that transfers funds from a savings account to a checking account. If the application successfully subtracts the funds from the savings account, but is interrupted by a system problem before it can add the funds to the checking account, the money would disappear, creating unbalanced bank accounts.

The two update commands (subtracting and adding funds) must be described as parts of a single logical transaction, so that the subtraction and addition updates are not written to the data source independently of each other.

Defining a Transaction

You define a logical transaction by issuing a COMMIT or ROLLBACK command following the last data source command of the transaction. For simplicity, the remainder of this topic refers to COMMIT only, but unless stated otherwise, both commands are meant. For example, the beginning of your application is the beginning of its first logical transaction. The data source commands that follow are part of the transaction. When the application issues its first COMMIT command, it marks the end of the first transaction.

The data source commands that follow the first COMMIT become part of the second logical transaction. The next COMMIT to be issued marks the end of the second transaction, and so on.

The COMMIT command defines the boundary of the transaction. All data source commands issued between two COMMIT commands are in the same transaction. This explanation describes the simplest case, in which a transaction exists entirely within a single procedure. When a transaction spans procedures, you have several options for deciding how to define a transaction boundary, as described in [When an Application Ends With an Open Transaction](#) on page 181.

Example: **Defining a Simple Transfer of Funds Transaction**

For example, transferring money from a savings account to a checking account requires two update commands. If you want to define the transfer, including both updates, as one logical transaction, you could use the following function:

```
CASE TransferMoney
  UPDATE Savings FROM SourceAccts
  UPDATE Checking FROM TargetAccts
  COMMIT
ENDCASE
```

When Does a Data Source Command Cause a Transaction to Fail?

A data source command can fail for many reasons. For example, an UPDATE command might try to write to a record that never existed because a key was mistyped, or an INCLUDE command might try to add a record that has already been added by another user.

In some cases, when a command fails, you might want to keep the transaction open and simply resolve the problem that caused the command to fail. For example, in the first case of attempting to update a record that does not exist, you might wish to ask the application user to correctly re-enter the customer code (which is being used as the key of the record). In other cases, you might wish to roll back the entire transaction.

If a data source command fails, it will only cause the logical transaction that contains it to be automatically rolled back in certain circumstances. The deciding factor is when a data source command fails. If a data source command fails when the transaction:

- Is open (that is, when the application issues the data source command), the transaction remains open, and the failed data source command does not become part of the transaction. This means that, if the application later attempts to commit the transaction, because the failed data source command is not part of the transaction, it will not affect the success or failure of the transaction.

You can evaluate the success of a data source command in an open transaction by testing the value of the FocError system variable immediately after issuing the command. If you wish the failure of the data source command to roll back the transaction, you must issue a ROLLBACK command.

- Is being closed (that is, when the application tries to commit the transaction), the failure of the data source command to be written to the data source causes the transaction to fail, and the entire transaction is automatically rolled back.

Canceling a Transaction

A transaction that is ongoing and has not yet been committed is called an open transaction. To cancel an open transaction, you must issue a ROLLBACK command. ROLLBACK voids any of the data source commands of the transaction that have already been issued so that none of them are written to the data source.

Transactions and Data Source Position

When a logical transaction is committed or rolled back, it resets all position markers in all the data sources that are accessed by the transaction procedures. Resetting the position markers points them to the beginning of the data source segment chains.

How Large Should a Transaction Be?

A transaction is at its optimal size when it includes only those data source commands that are mutually dependent upon each other for validity. If you include independent commands in the transaction and one of the independent commands fails when you try to commit the transaction, the dependent group of commands will be needlessly rolled back.

For example, in the following banking transaction that transfers funds from a savings account to a checking account, you should not add an INCLUDE command to create a new account, since the validity of transferring money from one account to another does not depend upon creating a new account.

```
CASE TransferMoney
  UPDATE Savings FROM SourceAccts
  UPDATE Checking FROM TargetAccts
  COMMIT
ENDCASE
```

Another reason for not extending transactions unnecessarily is that, in a multi-user environment, the longer a transaction takes, the more likely it is to compete for records with transactions submitted by other users. Transaction processing in a multi-user environment is described in [Concurrent Transaction Processing](#) on page 182.

Designing Transactions That Span Procedures

Logical transactions can span multiple Maintain Data procedures. If a Maintain Data procedure with an open transaction passes control to an App Studio procedure, the open transaction is suspended. When control next passes to a Maintain Data procedure, the transaction picks up from where it had left off.

When a transaction spans several procedures, you will usually find it easier to define the boundaries of the transaction if you commit it in the highest procedure in the transaction (that is, in the procedure closest to the root procedure). Committing a transaction in a descendant procedure of a complex application, where it is more difficult to track the flow of execution, makes it difficult to determine the transaction boundaries (that is, to know which data source commands are being included in the transaction).

When a child procedure returns control to its parent procedure, and the child has an open logical transaction, you have two options:

- ❑ You can continue the open transaction of a child into the parent procedure when the child returns control to the parent. Simply specify the KEEP option when you return control with the GOTO END command.
- ❑ You can close the open transaction of the child automatically at the end of the child procedure. By default, Maintain Data issues an implied COMMIT command to close the open transaction. You can also specify this behavior explicitly by coding the RESET option when you return control with the GOTO END command.

KEEP and RESET are described in [Command Reference](#) on page 51.

Example: Moving a Transaction Boundary Using GOTO END KEEP

Consider a situation where procedure A calls procedure B, and procedure B then calls procedure C. The entire application contains no COMMIT commands, so the initial logical transaction continues from the root procedure (A) through the descendant procedures (B and C). C and B both return control to their parent procedure using a GOTO END command.

The table below shows how specifying or omitting the KEEP option when procedures B and C return control affects the transaction boundaries of the application (that is, how the choice between KEEP and the implied COMMIT determines where the initial transaction ends, and how many transactions follow).

C returns to B with...	B returns to A with...	Transaction boundaries ()
KEEP	KEEP	A-B-C-B-A one transaction
KEEP	implied COMMIT	A-B-C-B A two transactions
implied COMMIT	KEEP	A-B-C B-A two transactions
implied COMMIT	implied COMMIT	A-B-C B A three transactions

Designing Transactions That Span Data Source Types

If a transaction writes to multiple types of data sources, each database management system (DBMS) evaluates its part of the transaction independently. When a COMMIT command ends the transaction, the success of the COMMIT against each data source type is independent of the success of the COMMIT against the other data source types. This is known as a broadcast commit. If any part of the broadcast commit fails, the value of FocCurrent is not zero (0).

For example, if you issue a Maintain Data procedure against the FOCUS data sources Employee and JobFile, and a DB2 data source named Salary, the success or failure of the COMMIT against Salary is independent of its success against Employee and JobFile. It is possible for it to be successful against Salary and write that part of the transaction, while being unsuccessful against Employee and JobFile and roll back that part of the transaction. Because it is unsuccessful against Employee and JobFile, the value of FocCurrent is not zero (0).

Designing Transactions in Multi-Server Applications

In an application that spans multiple WebFOCUS Servers, the server defines the maximum scope of a logical transaction. No transaction boundary can extend beyond a WebFOCUS Server. If one of your applications spans several servers, protect its transaction boundaries by ensuring that:

- ❑ All of the procedures of the application that read and write to a given data source reside on the same WebFOCUS Server.
- ❑ In each of the transactions of the application that span multiple procedures, all of the transaction procedures that read and write to data sources reside on the same WebFOCUS Server.

If a procedure with an open transaction calls another procedure that resides on a different WebFOCUS Server, and the situation violates either of the previous restrictions, the data source commands on the new server will comprise a new transaction. When control returns to the calling procedure on the original server, the original open transaction continues from where it had left off.

When an Application Ends With an Open Transaction

If an application terminates while a logical transaction is still open, Maintain Data issues an implied COMMIT command to close the open transaction, ensuring that any data source commands issued after the last explicit COMMIT are accounted for. The only exception is if your Maintain Data session abnormally terminates. Maintain Data does not issue the implied COMMIT, and any remaining uncommitted data source commands are rolled back.

Evaluating Whether a Transaction Was Successful

When you close a transaction by issuing a COMMIT or ROLLBACK command, you must determine whether the command was successful. If a COMMIT command is successful, then the transaction it closes has been successfully written to the data source. If a ROLLBACK command is successful, then the transaction it closes has been successfully rolled back.

The system variable FocCurrent provides the return code of the most recently issued COMMIT or ROLLBACK command. By testing the value of FocCurrent immediately following a COMMIT or ROLLBACK command, you can determine if the transaction was successfully committed or rolled back. If the value of FocCurrent is:

- ❑ Zero (0), the command was successful.
- ❑ Not zero (0), the command was unsuccessful.

FocCurrent is global to a procedure. If you want a given value of FocCurrent to be available in a different procedure, you must explicitly pass it as an argument to that procedure.

Example: Evaluating the Success of a Transaction

The following function commits a transaction to a data source. If the transaction is unsuccessful, the application invokes another function that writes to a log and then begins a new transaction. The FocCurrent line evaluates the success of the transaction:

```
CASE TransferMoney
  UPDATE AcctBalance FROM SourceAccts
  UPDATE AcctBalance FROM TargetAccts
  COMMIT
  IF FocCurrent NE 0 THEN PERFORM BadTransfer
ENDCASE
```

Concurrent Transaction Processing

Several applications or users often need to share the same data source. This sharing can lead to problems if they try to access a record concurrently, that is, if they try to process the same data source record at the same time.

To ensure the integrity of a data source, concurrent transactions must execute as if they were isolated from each other. The changes of one transaction to a data source must be concealed from all other transactions until that transaction is committed. To do otherwise, runs the risk of open transactions being exposed to interim inconsistent images of the data source, and consequently, corrupting the data source.

To prevent users from corrupting the data in this way, the database management system (DBMS) must coordinate concurrent access. There are many strategies for doing this. No matter which type of data source you use, Maintain Data respects your DBMS concurrency strategy and lets it coordinate access to its own data sources.

For more information about how your DBMS handles concurrent access, see your DBMS documentation. For FOCUS data sources, this information is presented in [Ensuring Transaction Integrity for FOCUS Data Sources](#) on page 184. For DB2, you can find some suggested strategies for writing Maintain Data transactions to DB2 data sources in [Ensuring Transaction Integrity for DB2 Data Sources](#) on page 191. For many other types of data sources, you can also apply the strategies described in [Ensuring Transaction Integrity for DB2 Data Sources](#) on page 191, changing DBMS-specific details when necessary.

Example: **Why Concurrent Access to a Data Source Must Be Managed Carefully**

Consider the following two applications that access the Employee data source:

- ❑ The Promotion application reads a list of employees who have received promotions, and updates their job codes to correspond to their new positions.
- ❑ The Salary application, run once at the beginning of each year, checks every employee job code and gives each employee an annual raise based on his or her job title. For example, assistant managers (job code A15) will earn \$30,000 in the new year, and managers (A16) will earn \$40,000.

Joan Irving is an assistant manager. Consider what happens when these two applications try to access and update the same record at the same time, without any coordination:

1. The Promotion application reads the record of Irving and, based on information in a transaction data source, indicates that she has been promoted to manager, and computes her new job code (A16).
2. The Salary application reads the record of Irving and, based on her job code in the data source (A15), computes her new salary (\$30,000).
3. The Promotion application writes the new job code (A16) to the data source.
4. The Salary application writes the new salary (\$30,000) to the data source.

Remember the earlier business rule (assistant managers earn \$30,000, managers earn \$40,000). Because two applications accessed the same record at the same time without any coordination, the rule has been broken (Joan Irving has a manager job code but the salary of an assistant manager). The data source has become internally inconsistent.

Ensuring Transaction Integrity for FOCUS Data Sources

Each database management system (DBMS) supports transaction integrity in its own way. The FOCUS DBMS manages concurrent access to FOCUS data sources using the FOCUS Database Server, and uses certain commands to identify transaction integrity attributes. The FOCUS Database Server was formerly known as a sink machine, or Simultaneous Usage (SU) facility on some platforms.

To ensure transaction integrity for FOCUS data sources, perform the following tasks:

- Install the FOCUS Database Server.** When installing each WebFOCUS Server that will host FOCUS data sources, select the FOCUS Database Server option.
- Set COMMIT.** Set the COMMIT server parameter to ON. This enables the COMMIT and ROLLBACK commands for FOCUS data sources, and enables the use of the FOCUS Database Server. For more information, see [Setting COMMIT](#) on page 184.
- Select which segments will be verified for changes.** Set the PATHCHECK server parameter to specify the type of segments for which the FOCUS Database Server will verify change. This is optional. You can accept the default setting. For more information, see [Selecting Which Segments Will Be Verified for Changes](#) on page 187.
- Identify the FOCUS Database Server.** Identify which FOCUS Database Server will manage concurrent access to each FOCUS data source. For more information, see [Identifying the FOCUS Database Server](#) on page 188.
- Start the FOCUS Database Server.** Under Windows and UNIX, when you start the WebFOCUS Server, it automatically starts the FOCUS Database Server. When you stop the WebFOCUS Server, it automatically stops the FOCUS Database Server. For information about starting and stopping the FOCUS Database Server under MVS and OS/390, see the *Simultaneous Usage Reference Manual, TSO Version* manual.

Setting COMMIT

You must set the COMMIT server parameter to ON before using the COMMIT and ROLLBACK commands for FOCUS data sources, and before using the FOCUS Database Server. You must set COMMIT on all WebFOCUS Servers hosting procedures that read or write to FOCUS data sources in a logical transaction. In most applications, this will mean setting COMMIT on all WebFOCUS Servers that host procedures with data source commands.

You can set COMMIT:

- Comprehensively for all users on a WebFOCUS Server. Issue the SET COMMIT command in the server global profile (EDASPROF).

- ❑ Comprehensively for a group of users on a WebFOCUS Server. Issue the SET COMMIT command in one or more of the group profiles of the server. Group profiles are supported under UNIX, OS/390, and MVS.
- ❑ Individually for each user on a WebFOCUS Server. Issue the SET COMMIT command in one or more of the group profiles of the server (PROFILE). The user in this case is the user account that launches the application.

If you set COMMIT in a user profile or group profile, you must set it in the profile of the user or group that runs the application.

You can also set COMMIT directly from a Maintain Data procedure.

Syntax: **How to Set COMMIT**

The COMMIT server parameter enables transaction integrity for FOCUS data sources. To set COMMIT, issue the SET COMMIT command in a WebFOCUS Server global profile, or in one or more of its user or group profiles, using the following syntax:

```
SET COMMIT={ON|OFF}
```

To set COMMIT in a Maintain Data procedure, use the following syntax

```
SYS_MGR.FOCSET("COMMIT" " {ON|OFF} ) "
```

where:

ON

Enables the COMMIT and ROLLBACK commands for use with FOCUS data sources, and enables the use of the FOCUS Database Server to ensure transaction integrity.

OFF

Disables the COMMIT and ROLLBACK commands for use with FOCUS data sources, and disables the use of the FOCUS Database Server to ensure transaction integrity. OFF is the default value.

Sharing Access to FOCUS Data Sources

The FOCUS DBMS ensures transaction integrity when multiple users are trying to access the same data source concurrently. If you are processing a transaction and, in the interval between beginning your transaction and completing it, the segments updated by your application have been changed and committed to the data source by another user, Maintain Data will roll back your transaction. This coordination is performed by the FOCUS Database Server. You can test if your transaction was rolled back by checking the value of the FocCurrent transaction variable, and then branch accordingly.

This strategy, where FOCUS verifies that the records to which you wish to write have not been written to by another user in the interim, is called change verification. It enables many users to share write access to a data source, and grants update privileges for a given record to the first user that attempts the update.

Change verification takes advantage of the fact that two users rarely try to update the same record at the same time. Some DBMSs use strategies that lock out all but one user. Others grant update privileges to the first user that retrieves a record, even if he or she is the last one ready to update it, resulting in a performance bottleneck. In contrast, the FOCUS DBMS strategy of change verification enables the maximum number of users to access the same data concurrently, and makes it possible to write the maximum number of transactions in the shortest time. The FOCUS Database Server and change verification strategy are designed for high-performance transaction processing.

How the FOCUS Database Server and Change Verification Work

The change verification strategy of the FOCUS Database Server is an extension of basic transaction processing. Each application user who accesses the FOCUS Database Server is known as a client. To ensure transaction integrity follow this simple change verify protocol:

1. As always, use the NEXT or MATCH commands to retrieve the data source records you need for the current transaction. When the application issues these commands, the server sends the application a private client copy of the records.

Note: Do not retrieve data from a data source by running a report procedure. The FOCUS Database Server does not check this data for changes when you attempt to commit a transaction.

2. When the application issues a data source write command (such as INCLUDE, UPDATE, REVISE, or DELETE) against the retrieved records, it updates its private copy of the records.
3. When the application issues a COMMIT command to indicate the end of the transaction, the application session sends a log of the transaction back to the server. The server now checks to see if any of the segments that the transaction changed have, in the interim, been changed and committed to the data source by other clients, and if any segments that the transaction added have, in the interim, been added by other clients. You can customize which segments the FOCUS Database Server checks for changes by setting the PATHCHECK server parameter, as described in [Selecting Which Segments Will Be Verified for Changes](#) on page 187.

The server takes one of the following actions:

- No conflict.** If none of the records have been changed or added in the interim, then the transaction is consistent with the current state of the data source. The server writes the transaction to the data source and sets the FocCurrent transaction variable of the application to zero (0) to confirm the update.

- ❑ **Conflict.** If any records have been changed in the interim, then the transaction might be inconsistent with the current state of the data source. The server ignores the transaction changes to the data source, rolls back the transaction, and alerts the application by setting FocCurrent to a non-zero (0) number.
4. The application evaluates FocCurrent and branches to the appropriate function.

Selecting Which Segments Will Be Verified for Changes

When you use a FOCUS Database Server, you can customize the change verification process by defining the segments for which the FOCUS Database Server will verify changes. You define this, using the server parameter described in [How to Set PATHCHECK](#) on page 187.

You can choose between:

- ❑ **All segments in the path.** The FOCUS Database Server verifies that all segments in the path extending from the root segment to the target segment have not been changed and committed in the interim by other users.
- ❑ **Modified segments only.** The FOCUS Database Server determines which segments you are updating or deleting, and verifies that those segments have not been changed and committed in the interim by other users.

You can set PATHCHECK for each FOCUS Database Server, which affects all applications that access FOCUS data sources managed by that FOCUS Database Server. To set it under:

- ❑ **Windows and UNIX,** issue the SET PATHCHECK command in the batch file (EDASTART.BAT) that starts the FOCUS Database Server.
- ❑ **OS/390 and MVS,** issue the SET PATHCHECK command in the FOCUS Database Server profile (HLIPROF).

Syntax: **How to Set PATHCHECK**

The PATHCHECK server parameter defines which segments the FOCUS Database Server will check for changes. To set PATHCHECK, issue the SET PATHCHECK command in the batch file that starts the FOCUS Database Server, using the following syntax

```
SET PATHCHECK={ON|OFF}
```

where:

ON

Instructs the FOCUS Database Server to verify that all segments in the path extending from the root segment to the target segment have not been changed and committed in the interim by other users. This is the default for OS/390 and MVS.

OFF

Instructs the FOCUS Database Server to check only segments that the current transaction has updated or deleted, and verify that those segments have not been changed and committed in the interim by other users. This is the default for Windows and UNIX.

Identifying the FOCUS Database Server

To identify which FOCUS Database Server will manage access to a given FOCUS data source, you must issue a command that associates the server with the data source, as described in [How to Identify a FOCUS Database Server With USE](#) on page 188.

You can issue this command:

- Comprehensively, for all users on a WebFOCUS Server. Issue the USE command in the server profile (EDASPROF).
- Individually, for each user on a WebFOCUS Server. Issue the USE command in the user server profile (PROFILE). The user in this case is the user account that launches the application.

Syntax: How to Identify a FOCUS Database Server With USE

For each FOCUS data source that will be managed by a FOCUS Database Server, you must associate the data source with the server by issuing a USE command in a WebFOCUS Server profile. The USE command syntax is:

```
USE
datafile ON server_id
[datafile ON server_id]
.
.
.
END
```

where:

datafile

Is the file specification of a data source to be managed by the FOCUS Database Server.

server_id

Under Windows and UNIX, is the node name of the FOCUS Database Server, as defined in the FOCUS Database Server node block of the Data Server configuration file.

Under OS/390 and MVS, is the ddname of the communication dataset that points to the FOCUS Database Server job.

If you wish, you can identify multiple data source and server pairs in one USE command.

Using Report Procedures and a FOCUS Database Server

When a FOCUS Database Server manages access to a FOCUS data source, each logical transaction that accesses that data source works with its own private copy of the data source records. This ensures that the transaction sees a consistent image of the data source that is isolated from changes being attempted by other users.

App Studio procedures, such as report procedures, are not part of a logical transaction. When control passes from a Maintain Data procedure to an App Studio procedure, the open transaction is suspended for the duration of the App Studio procedure. Therefore, if the App Studio procedure reports against a FOCUS data source, it accesses the live data source. It does not open the private copy of the transaction. Changes made by the open transaction are not seen by the report, and changes committed by other users since the open transaction began are seen by the report, though not necessarily by the open transaction.

For similar reasons, you should not use a report procedure to retrieve data for use in a transaction. The FOCUS Database Server does not check this data for changes when you attempt to commit a transaction. Always use the NEXT or MATCH commands to retrieve transaction data.

If you wish to deploy App Studio procedures containing report requests to a WebFOCUS Server that also hosts Maintain Data procedures, you must represent the server as two different outbound nodes, and deploy App Studio reporting procedures to one node and Maintain Data procedures to the other node, as described in [Accessing Report Procedures When Using a FOCUS Database Server](#) on page 190. Otherwise, the App Studio procedures may interfere with your transaction logic.

Accessing Report Procedures When Using a FOCUS Database Server

If you are using a FOCUS Database Server and you wish to access Maintain Data procedures and App Studio report procedures that are located on the same WebFOCUS Server (referred to here as the target server), you must:

1. Represent the target server as two different outbound nodes:
 - ❑ Each WebFOCUS Server that executes Maintain Data procedures or App Studio procedures that are located on the target server.

This requirement also applies to the target server itself. If it executes Maintain Data procedures or App Studio procedures that are deployed on itself, it must be represented to itself as two outbound nodes.

2. Of the procedures that you will be deploying to the target server, deploy the App Studio report procedures to one of these outbound nodes, and deploy the Maintain Data procedures to the other outbound node.

This is necessary because deploying both types of procedures to the same outbound node can cause report logic to corrupt transaction integrity.

***Procedure:* How to Access Maintain Data and App Studio Report Procedures on the Same Server**

In an application that uses a FOCUS Database Server, if you want to deploy to the Maintain Data procedures of the application and App Studio report procedures on the same WebFOCUS Server (referred to here as the target server):

1. Represent the target server as two outbound nodes that have different remote server names but the same protocol options (for the TCP/IP protocol, this means specifying the two nodes with the same IP address, port number, and compression setting).
2. Represent the target server to WebFOCUS Servers as two outbound nodes. Perform this step for each WebFOCUS Server that executes Maintain Data procedures or App Studio procedures that are deployed on the target server. This requirement also applies to the target server itself. If it executes Maintain Data procedures or App Studio procedures that are deployed on itself, it must be represented to itself as two outbound nodes.

As in step 1, define the two outbound nodes as having different remote server names but the same protocol options (for the TCP/IP protocol, this means specifying the same IP address, port number, and compression setting). You can do this by copying the target server node block, as described in the WebFOCUS Server [Communications Configuration File Location and Name](#) on page 191, and paste it just below the end of the original block. The node block begins with the NODE keyword and continues through the END keyword. Edit the pasted block to provide a new eight-character node name, but leave the other values unchanged.

If the WebFOCUS Server is the target server, then copy the target server node block from the App Studio `odin.cfg` file and paste it twice, into the communications configuration file of the target server. Edit the second pasted block to provide a new eight-character node name, but leave the other values of the block unchanged.

For information about the name and location of the communications configuration server file, see [Communications Configuration File Location and Name](#) on page 191.

3. Deploy the application.

Reference: Communications Configuration File Location and Name

The WebFOCUS Server communications configuration file under:

- ❑ **Windows and UNIX** is `odin.cfg`, and resides in the `etc` subdirectory of the Data Server configuration directory. Under UNIX, `odin.cfg` is in the `$EDACONF/etc` directory. Under Windows, you can find the name of the configuration directory in the environment variable `EDACONF`.

When you install and configure a WebFOCUS Server, the configuration directory defaults to `$HOME/ibi/srv82/server_instance` under UNIX, and `home\ibi\srv82\server_instance` under Windows. `HOME` is an environment variable whose value in this context is the name of the home directory of the user account that installed the WebFOCUS Server. `server_instance` is the name of a server configuration directory (there is one directory per server instance. The convention is to name this directory `wfs` for a WebFOCUS Server, and `wfm` for a Maintain Data Application Server).

- ❑ **OS/390 and MVS** is allocated to `ddname EDACSG` in the server startup JCL.

Sharing Data Sources With Legacy MODIFY Applications

A FOCUS data source being managed by a FOCUS Database Server can be accessed by both Maintain Data applications and legacy MODIFY applications. Note that while MODIFY allows creating records with duplicate keys, Maintain Data does not support FOCUS data sources that have duplicate keys.

Ensuring Transaction Integrity for DB2 Data Sources

DB2 ensures transaction integrity by locking data source rows when they are read. The behavior of a lock depends on the isolation level of a transaction. The techniques suggested here for Maintain Data applications all use an isolation level of repeatable read. Repeatable read involves a trade-off. It ensures absolute transaction integrity, but it can prevent other users from accessing a row for long periods of time, creating performance bottlenecks.

Under repeatable read, a row is locked when it is retrieved from the data source, and is released when the transaction that retrieved the row is either committed to the data source or rolled back. A Maintain Data DB2 transaction is committed or rolled back each time a Maintain Data application issues a COMMIT or ROLLBACK command. You explicitly code COMMIT and ROLLBACK commands in your Maintain Data application. In some circumstances the application may also issue these commands implicitly, as described in [Designing Transactions That Span Procedures](#) on page 179, and in [When an Application Ends With an Open Transaction](#) on page 181.

We recommend two strategies for writing transactions to DB2 data sources:

- ❑ **Using transaction locking to manage DB2 row locks.** This locks each row for the duration of the transaction, from the time a row is retrieved, until the transaction is committed. In effect, it relies on DB2 to ensure transaction integrity. This is simpler to code, but keeps rows locked for a longer period of time. This is the preferred strategy, unless the duration of its locks interferes excessively with your data source concurrency requirements.
- ❑ **Using change verification to manage DB2 row locks.** This locks each row while it is being retrieved, releases the lock, and then relocks the row shortly before writing it to the data source. This technique ensures transaction integrity by verifying, before writing each row, that the row has not been changed by other users in the interim. This is more complex to code, but locks rows for a shorter period of time, increasing data availability.

While these strategies are described for use with DB2 data sources, you can also apply them to transactions against other kinds of data sources, changing DBMS-specific details when necessary.

Reference: How Maintain Data DB2 Logic Differs From Other Information Builders Products

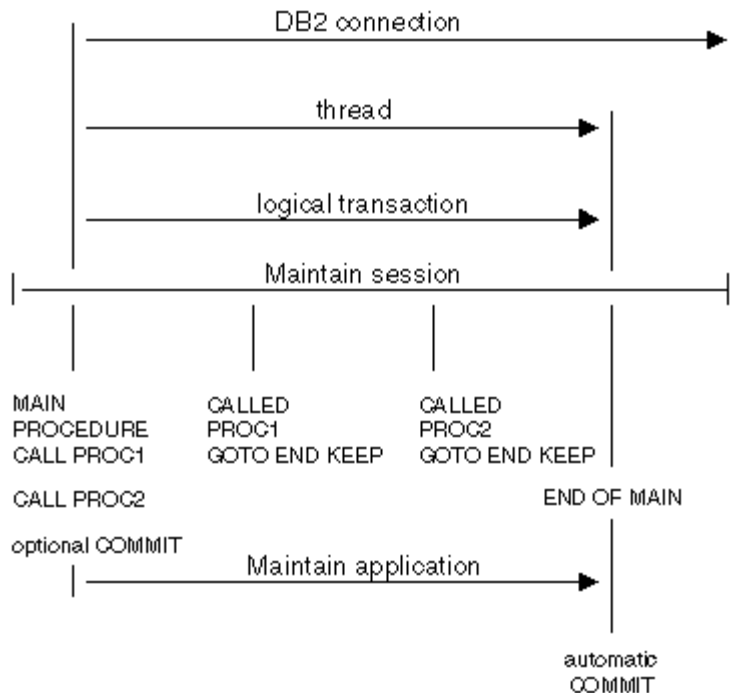
If you are familiar with using the Data Adapter for DB2 with Information Builders products other than Maintain Data, note that Maintain Data works with DB2 a bit differently:

- ❑ Maintain Data enables you to issue COMMIT and ROLLBACK commands explicitly. It also issues them implicitly in certain situations, as described in [Designing Transactions That Span Procedures](#) on page 179, and in [When an Application Ends With an Open Transaction](#) on page 181.
- ❑ Maintain Data does not support the SQL DB2 SET AUTOCOMMIT command to control automatic commits.
- ❑ Because Maintain Data works on sets of rows, the Data Adapter for DB2 does not automatically generate change verification logic.

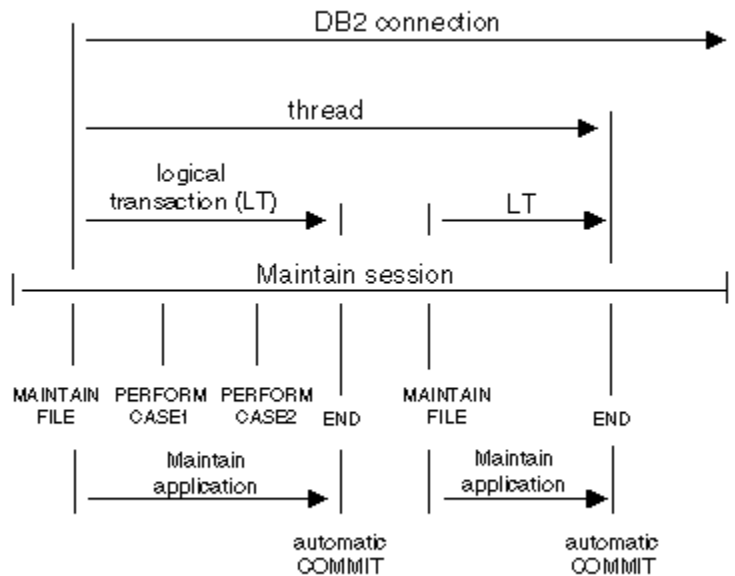
Using Transaction Locking to Manage DB2 Row Locks

You can use the transaction locking strategy to manage DB2 row locks in Maintain Data applications. While this strategy is described for use with DB2 data sources, you can also apply it to transactions against other kinds of data sources, changing DBMS-specific details when necessary. When using transaction locking, your application locks each row with an isolation level of repeatable read for the duration of the transaction, from the time it retrieves the row, until the time it commits or rolls back the transaction.

The following illustration shows the duration of connections, threads, and logical transactions (also known as logical units of work) when you use this strategy.



If your applications are small in scope, comprising only a single procedure, the duration of connections, threads, and logical transactions would look like the following illustration:



Compared to change verification, transaction locking is simpler to code, but keeps rows locked for a longer period of time. This may cause other users to experience time outs, in which case DB2 will return a -911 or -904 SQL code. You can mitigate the effect of row locking by:

- Keeping the size of the transaction small, making it less likely that another user will encounter a row locked by your transaction.
- Implementing the change verification strategy described in [Using Change Verification to Manage DB2 Row Locks](#) on page 195.
- Having user applications check for a locked condition when retrieving rows, and upon encountering a lock, re-issuing the retrieval request a specified number of times in a loop. If the user application exceeds the specified number of attempts, have it display a message to the user indicating that the row is in use, and suggesting that the user try again later.
- Using standard database administration techniques, such as report scheduling, tablespace management, and data warehousing.

Procedure: How to Implement Transaction Locking for DB2

To implement the transaction locking strategy for managing DB2 row locks in Maintain Data applications, bind the Data Adapter for DB2 plan with an isolation level of repeatable read. The isolation level is a Data Adapter for DB2 installation BIND PLAN parameter. In your Maintain Data application:

1. **Read the rows.** Retrieve all required rows. Retrieval locks the rows with an isolation level of repeatable read.
2. **Write the transaction to the data source.** Apply the updates of the transaction to the data source.
3. **Be sure to terminate called procedures correctly.** If a Maintain Data procedure calls another Maintain Data procedure within the scope of a transaction, the called procedure must return control using the GOTO END KEEP command. For more information about GOTO END KEEP, see [Designing Transactions That Span Procedures](#) on page 179.

Caution: If any called procedure within the scope of a transaction returns control without GOTO END KEEP, Maintain Data issues an implied COMMIT command, releasing all row locks and making the application vulnerable to updates by other users. Be sure to return control using GOTO END KEEP. Otherwise, code each transaction within a single procedure, so that the scope of each transaction does not extend beyond one procedure, or use the change verification strategy described in [Using Change Verification to Manage DB2 Row Locks](#) on page 195.

4. **Close the transaction.** When the transaction is complete, close it by issuing a COMMIT or ROLLBACK command. The COMMIT or ROLLBACK command releases all row locks.

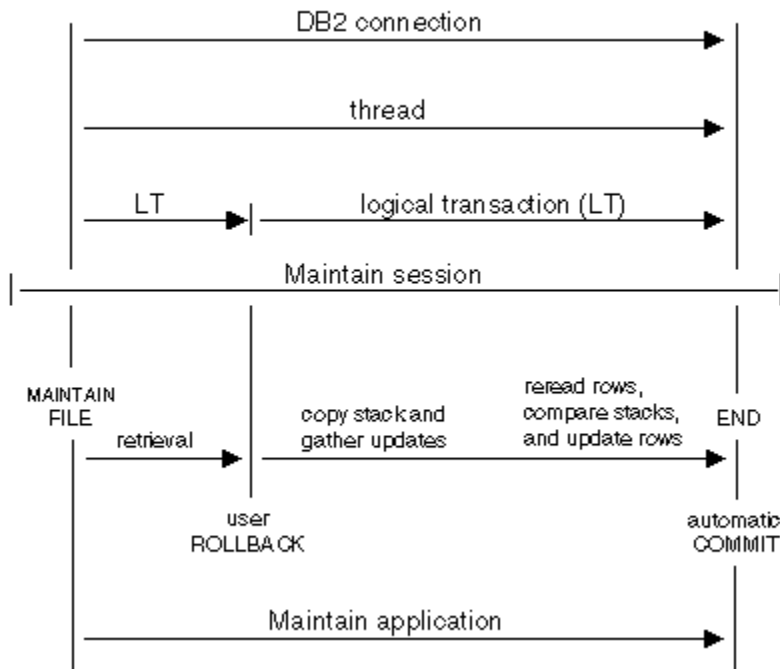
Using Change Verification to Manage DB2 Row Locks

You can use the change verification strategy to manage DB2 row locks in Maintain Data applications. While this strategy is described for use with DB2 data sources, you can also apply it to transactions against other kinds of data sources by changing DBMS-specific details when necessary.

When using change verification, your application retrieves all needed rows into a stack, locking them in the process, releases the locks after retrieval, and then performs all updates against the stack (not against the data source). This enables you to work with the data in the stack as long as necessary without preventing other users from accessing the data source. When you are ready to close the transaction, retrieve the original rows from the data source again, relocking them in the process. Then, compare their current values in the data source to their original values when you first retrieved them, and write the transaction to the data source if the values are the same, that is, if the rows have not been changed by other users in the interim.

Change verification enables the maximum number of users to access the same data concurrently, and makes it possible to write the maximum number of transactions in the shortest time. It is able to do this because it is an optimistic locking protocol, that is, it is optimized for the most common situation, in which at any moment, at most one user will attempt to update a given row. Compared to transaction locking, this is more complex to code, but locks rows for less time, increasing data availability.

The following illustration shows the duration of connections, threads, and logical transactions when you use this strategy for DB2 data sources.



Procedure: How to Implement Change Verification for DB2

To implement the change verification strategy for managing DB2 row locks in Maintain Data applications, bind the Data Adapter for DB2 plan with an isolation level of repeatable read. The isolation level is a Data Adapter for DB2 installation BIND PLAN parameter. In your Maintain Data application:

1. **Read the rows.** Retrieve all required rows into a stack (for example, Stack1). Retrieval locks the rows with an isolation level of repeatable read.

2. **Free the row locks.** Issue a ROLLBACK command immediately following retrieval in order to release all row locks.
3. **Copy the stack.** Make a copy of the stack (for example, Stack2). You will use this copy later when checking for changes.
4. **Write the transaction to the stack.** Apply the updates of the transaction to the rows in the original stack (Stack1).
5. **Read the rows again.** Retrieve the rows of the transaction from the data source into a new stack (for example, Stack3). Retrieval relocks the rows with an isolation level of repeatable read.
6. **Verify changes.** Compare the original data source values in the copy of the original stack (Stack2) to the current data source values (Stack3) to verify that other users have not changed these rows in the interim.
7. **Write the transaction to the data source.** If any of these rows have been changed in the data source by another user, you can roll back the transaction or take some other action, as your application logic requires. If none of the rows in the transaction have been changed by other users in the interim, your application can apply the transaction updates to the data source, and issue a COMMIT command to commit the transaction.

The COMMIT or ROLLBACK command releases all row locks.

Developing Classes and Objects

Most application development is modular. The developer creates complex systems comprised of smaller parts. In procedural development, these modules are procedures, and data is defined within each procedure. In object-oriented development, the modules are models of real-world objects (such as a customer or a shipping order), and both data and procedures are defined within each object. The object encapsulates the data and the procedures.

For example, if you are developing an order fulfillment system for a mail-order clothing business, the objects might include customers, orders, and stock items. The data of a customer object might include the customer ID code, phone number, and order history. The customer processes might include functions that add the customer to a new mailing list, update the customer information, and place an order for the customer.

Object-oriented development is a more efficient way of developing applications because it models the real-world objects with which your enterprise deals, and encourages you to reuse application logic in a variety of ways. You can use App Studio Maintain to create applications using object-oriented development, procedural development, or a hybrid of these two methods, providing you with a flexible development path.

In this appendix:

- [What Are Classes and Objects?](#)
 - [Defining Classes](#)
 - [Reusing Classes: Class Libraries](#)
 - [Declaring Objects](#)
-

What Are Classes and Objects?

Most applications need many objects of the same type. For example, if your business has 500 customers, you need one object to represent each customer. No one would want to design a customer object 500 times. Clearly, you need a template that defines all customer objects, so that you can design the template once, and use it often. For example, you would use the template each time you create a new customer object to represent a new customer.

An object template is called its class. Each object is an instance of a class. In other words, the class defines the type of object. In fact, when you create a class, the class becomes a new data type. Just as you can use a built-in data type, such as integer or alphanumeric, to define a simple variable, you can use a class data type to define an object.

Unlike a Master File, which is also a kind of template, a class defines both variables and functions. Just as a built-in data type defines the operations that you can perform on data of that type (for example, you can perform addition, subtraction, division, and multiplication on integers), a class defines the functions that you can perform on objects of that class (for example, you can invoke functions to update an address and to place an order for a customer object).

***Example:* Comparing Classes and Built-in Data Types**

Just as you can use the alphanumeric built-in data type to define a customer ID code as an A8 variable

```
DECLARE CustID/A8;
```

you can use the RetailCustomer class to define a customer as an object:

```
DECLARE CustSmit8942/RetailCustomer;
```

Class Properties: Member Variables and Member Functions

You define a class by describing its properties. Classes have two kinds of properties:

- ❑ Data, in the form of the variables of a class. Because these variables exist only as members of the class, they are called member variables. In some object-oriented development environments these are also known as object attributes or instance variables.

Class member variables determine what the class is (as opposed to what it does). Each object of that class can have different values for its member variables.

- ❑ Processes, implemented as functions. Because these functions exist only as members of the class, they are called member functions. In some object-oriented development environments, these are also known as methods.

The member function of a class defines its behavior. They determine what you can do to objects of that class, and in what ways you can manipulate the data.

***Example:* Member Variables for a Customer Class**

An application for a mail-order clothing business has defined a customer class named Customer. The member variables of the class might include the customer code, name, phone number, and most recent order number of the customer:


```

DESCRIBE Customer =
(IDcode/A6,
 LastName/A15,
 FirstName/A10,
 Phone/I10,
 LastOrder/A15);
.
.
.
ENDDESCRIBE

```

After declaring a new customer object for the customer, Frances Smith

```

DECLARE CustFrSmith/Customer;

```

you can assign a value to the IDcode member variable:

```

DECLARE CustFrSmith.IDcode = GetNewCustCode();

```

Each object can have different values for its member variables. For example, in this case, each customer will have a different ID code.

***Example:* Member Functions for a Customer Class**

An application for a mail-order clothing business has defined a customer class named Customer. The class member functions might include a function that adds the customer to a new mailing list, a function that updates the customer contact information, and a function that places an order for the customer:

```

DESCRIBE Customer =
(IDcode/A6,
 Phone/I10,
.
.
.
LastOrder/A15);
CASE AddToList TAKES Name/A25, Address/A50, IDcode/A6;
.
.
.
ENDCASE
CASE UpdateContact ...

CASE PlaceOrder ...

ENDDESCRIBE

```

After declaring a new customer object for the customer Frances Smith

```

DECLARE CustFrSmith/Customer;

```

you could add Frances Smith to the mailing list, using the AddToList member function:

```
CustFrSmith.AddToList();
```

Each object has the same member functions, and therefore, the same behavior. In this case, for example, each customer will be added to the mailing list using the function.

Inheritance: Superclasses and Subclasses

If you want to create a new class that is a special case of an existing class, you could derive it from the existing class. For example, in a human resources application, a class called Manager could be considered a special case of a more general class called Employee. All managers are employees, and possess all employee attributes, plus some additional attributes unique to managers. The Manager class is derived from the Employee class, so Manager is a subclass of Employee, and Employee is the superclass of Manager.

A subclass inherits all of its superclass properties. For example, it inherits all of the member variables and member functions of the superclass. When you define a subclass, you can choose to override some of the inherited member functions, meaning that you can recode them to suit the ways in which the subclass differs from the superclass. You can also add new member functions and member variables that are unique to the subclass.

Defining Classes


Before you can declare an object (an instance of a class), your procedure must have a class definition for that type of object. If the class:

- Is already defined in a class library, simply import the library into your procedure. Class libraries, which are implemented as import modules, are described in [Reusing Classes: Class Libraries](#) on page 209.
- Is already defined in another procedure, simply copy and paste the definition into a class library. You can then import the library into any procedure that needs it.
- Is not yet defined anywhere, you can define it in a class library or procedure using the Class Editor, or by coding the definition directly in the Procedure Editor, using the DESCRIBE command. If you define it in a class library, you can use the class definition in multiple procedures by simply importing the library into those procedures.

Procedure: How to Define a Class Using the Class Editor

This procedure describes how to define a new class. If you wish to define a new subclass, that is, a class that inherits properties from another class, see [How to Define a Subclass Using the Class Editor](#) on page 203.


1. In the Requests & Data Sources panel, right-click the import module or procedure and select *New class* in the shortcut menu.
2. In the New Class dialog box, type a name for your class.
3. Click the *Variables* tab to specify the member variables of a class. The member variables of a class express its properties.

4. To add a variable, click the *New* button .

The Member Variable dialog box opens.

The name of each member variable must be unique within the class to which it belongs. It can be identical, however, to the names of member variables of other classes.

5. Repeat step 4 to create any additional variables.
6. Click the *Functions* tab to specify the member functions of a class. The member functions of a class define the actions that can be performed on the objects of a class.

7. To add a function, click the *New* button .

The Member Function dialog box opens.

The name of each member function must be unique within the class to which it belongs. It can be identical, however, to the names of member functions of other classes.


8. Repeat step 7 to create any additional functions.
9. Optionally, click the *Description* tab and add a description to your class.
10. Click *OK* to confirm the class definition.

Procedure: How to Define a Subclass Using the Class Editor

To define a new class (a subclass) by inheriting properties (member functions and member variables) from another class (a superclass):

1. In the Requests & Data Sources panel, right-click the import module or procedure and select *Class (Describe)* in the submenu.
2. In the New Class dialog box, type a name for your class.

3. Select the class whose properties the new class will inherit from the *Inherits behavior from* list. You can choose from all of the classes that are defined in this import module or procedure, and in any modules that have been imported into it. The selected class will be the superclass, and the new class will be the subclass.
4. Click the *Variables* tab to specify the member variables of a class. The member variables of a class express its properties. A subclass inherits all the member variables of its superclass, and you can add new ones.


5. To add a variable, click the *New* button .

The Member Variable dialog box opens.

Note: You cannot delete member variables inherited from the superclass.

6. Repeat step 5 to create any additional variables.
7. Click the *Functions* tab to specify the member functions of a class. The member functions of a class define the actions that can be performed on the objects of a class.

Note that you cannot delete member functions inherited from the superclass. However, you can override an inherited member function to edit or remove its source code.

8. To add a function, click the *New* button .

The Member Function dialog box opens.

The name of each member function must be unique within the class to which it belongs. It can be identical, however, to the names of member functions of other classes.

9. Repeat step 8 to create any additional functions.
10. Optionally, click the *Description* tab and add a description of the class.
11. Click *OK* to confirm the class definition.

Syntax: How to Define a Class or Subclass Using the DESCRIBE Command

When you define a class using the Class Editor, it generates the definition in the procedure as a DESCRIBE command. If you wish to work directly with source code, you can create new class definitions and edit existing definitions directly in the Procedure Editor by using the following DESCRIBE syntax. You must issue the DESCRIBE command outside of a function, for example, at the beginning of the procedure prior to all functions.

```
DESCRIBE classname = ([superclass +] memvar/type [, memvar/type] ...)  
[ ; ]  
[ memfunction  
[ memfunction ] ...  
ENDDDESCRIBE ]
```

where:

classname

Is the name of the class that you are defining. The name is subject to the standard naming rules of the Maintain Data language.

superclass

Is the name of the superclass from which you wish to derive this class. Include only if this definition is to define a subclass.

memvar

Names one of the member variables of the class. The name is subject to the Maintain Data language standard naming rules.

type

Is a data type (a built-in format or a class).

memfunction

Defines one of the class member functions. Member functions are defined the same way as other Maintain Data functions, using the CASE command.

;

Terminates the definition if the definition omits member functions. If it includes member functions, the semicolon (;) is omitted and the ENDDDESCRIBE command is required.

ENDDDESCRIBE

Ends the class definition if it includes member functions. If it omits member functions, the ENDDDESCRIBE command must also be omitted, and the definition must be terminated with a semicolon (;).

Procedure: How to Edit a Class Definition

To add a new member function or member variable:

1. In the Requests & Data Sources panel, right-click the class and select *New member*, and then select *Function* or *Variable*.
2. In the New Function or New Variable dialog box, create your new function or variable.

To edit one of the member functions or member variables of a class:

1. In the Requests & Data Sources panel, right-click one of the class member functions or member variables.
2. In the shortcut menu, click *Edit*.

3. Make any necessary changes to the class definition in the Edit Variable or Member Function dialog boxes. For general information about editing a class definition, see [Defining Classes](#) on page 202.
4. Click *OK* to confirm your changes.

***Procedure:* How to Edit the Class Source Code**

If you wish, you can edit the Maintain Data code directly in the Procedure Editor.

1. In the Requests & Data Sources panel, right-click the class.
2. In the shortcut menu, click *Edit source*.
3. Make any changes you wish to the code between `DESCRIBE classname` and `ENDDESCRIBE`.
4. Close the Procedure Editor.

***Procedure:* How to Rename a Class, Member Variable, or Member Function**

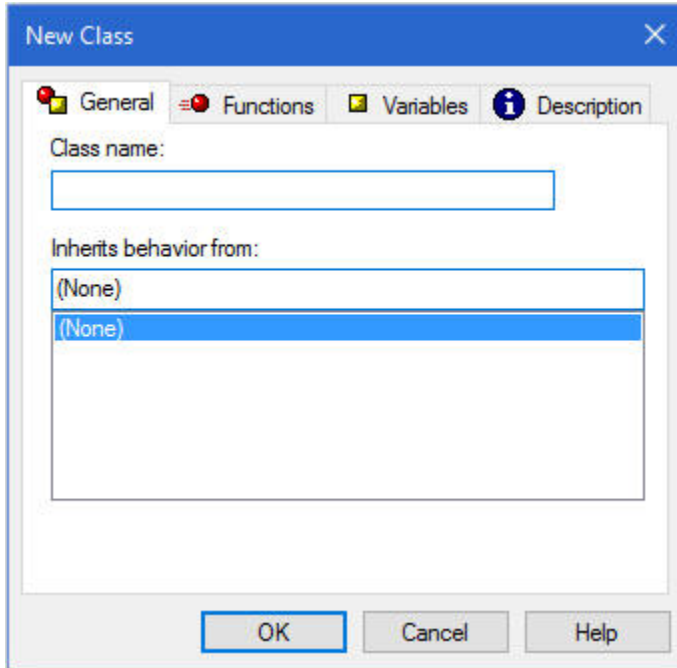
1. In the Requests & Data Sources panel, right-click the class, member variable, or member function, and in the shortcut menu, click *Rename*.
2. Type the new name.
3. Press the Enter key to confirm the new name.

***Procedure:* How to Delete a Class, Member Variable, or Member Function**

In the Requests & Data Sources panel, right-click the class, member variable, or member function, and in the shortcut menu, click *Delete*.

Reference: New Class and Edit Class Dialog Boxes

The New Class and Edit Class dialog boxes enable you to create and edit classes. An example of the New Class dialog box is shown in the following image.



The New Class dialog box contains the following tabs:

General tab

The General tab has the following options:

Class name

Type the name of your class here.

Inherits behavior from

If this class is based on another class, select that class from the list. The list comprises all of the classes that are defined in the open import module or procedure, and in any modules that have been imported into it.

Functions tab

The Functions tab includes the following options:

Member Functions

Lists the names of the member functions of a class.



Opens the Member Function dialog box, where you can define a new function.



Deletes a selected function from the list of functions, or if a function has been overridden, deletes the override.



Moves a selected function up in the list of functions.



Moves a selected function down in the list of functions.

Variables tab

The Variables tab includes the following options:

Member Variables: (Name/Type)

Lists the class member variables, including those that it has inherited.



Opens the Member Variable dialog box, where you can define a new variable.



Deletes a selected variable from the list of variables. You cannot delete variables that are inherited from another class definition.



Moves a selected variable up in the list of variables.



Moves a selected variable down in the list of variables.

Description tab

The Descriptions tab has the following elements:

Edit box

You can document the class by typing a description of it in this edit box. Maintain Data will turn your description into a comment in the class definition source code in the procedure or module.

Reusing Classes: Class Libraries

You can define a class once, but use it in multiple Maintain Data procedures by storing its definition in a class library. Libraries are a very useful way of reusing source code, enabling you to develop applications more efficiently.

A class library is implemented as an import module (a kind of non-executable procedure) in which you can store class definitions, as well as Maintain Data functions. After you have created a module, you can import it into each Maintain Data procedure in which you want to use those classes.

After you have created the import module, simply create new class definitions in the module, or copy existing definitions into the module. You create and edit class definitions in a module in the same way that you create and edit them in a procedure, as described in [Defining Classes](#) on page 202.

You can nest modules to any depth. For example, if you have two import modules named ClasLib1 and ClasLib2, you can import ClasLib1 into ClasLib2.

Note: A library cannot contain an explicit Top function, and cannot refer to data sources. For example, class definitions in a library cannot contain data source commands (such as NEXT and INCLUDE) and cannot refer to data source stacks.

Syntax: How to Import a Class Library Using the MODULE IMPORT Command

You can use the MODULE command to import libraries containing class definitions so that the current procedure can use those classes. Libraries can also contain other source code, such as function definitions. The syntax is

```
MODULE IMPORT (library_name [, library_name] ... );
```

where:

library_name

Is the name of the Maintain Data procedure that you wish to import as a source code library. Specify its file name without an extension. The file must reside in the path defined by the EDASYNR environment variable.

The MODULE command must immediately follow the MAINTAIN command.

Declaring Objects

After a class definition exists, you can declare objects of that class. This is identical to declaring simple variables of a built-in data type. You can declare objects using the Class Editor, or by coding the declaration directly in the Procedure Editor.

Procedure: How to Declare an Object Using the Variable Editor


Prerequisite: When declaring an object (that is, a class instance), the procedure in which you are declaring it must already include or import the class definition.

To declare an object using the Variable Editor:

1. Select the procedure in which you want the object to be declared.
2. Right-click the procedure, click *New* in the shortcut menu, and click *Variable (Declare)* in the submenu.

or

Click the *New variable* button  on the Application toolbar.

3. In the New Variable dialog box, type a name for your object in the *Name* field.
4. Open the Type drop-down combo box. If the class of which this object will be an instance:
 - Is listed in the Type combo box, select it, and skip to step 9.
 - Is not listed in the Type combo box, and you know its name, enter the name in the Type combo box and skip to step 9.
 - Is not listed in the Type combo box, and you wish to select it from a list of all the class procedures, click the ellipsis button  to open the Type Wizard and continue with step 5.
5. Select *User-defined Class* from the drop-down list.
6. From the drop-down list select *Simple* for a single object, or *Stack of* for a list of objects.

7. Select a class from the list.

This adds the class to the list of data types in the Type combo box in the New Variable dialog box, making it available to you when you create additional objects in the future.

8. Click *OK* to return to the New Variable dialog box.
9. Optionally, click the *Description* tab and enter a description. This description will be generated as a comment with the object declaration's source code.
10. Click *OK* to confirm the object declaration.
11. The Variable Editor created a *global* object declaration. If you wish to convert this to a local object declaration, while in the Procedure Editor simply cut the declaration from the \$ \$Declarations section at the top of the procedure, and paste the declaration to the desired function.

Note that local declarations must immediately follow the function CASE command, preceding all the other commands in the function.

Syntax: How to Declare an Object Using the DECLARE Command

You can declare a local or global object in the Procedure Editor using the DECLARE command. To make the declaration:

- Local, code the DECLARE command in the function to which you want it to be local, following the function CASE command, and preceding all the other commands in the function.
- Global, code the DECLARE command outside of any function. It is recommended that you use the \$ \$Declarations section at the top of the procedure to make the declaration easier for you to find.

You can also create global objects using the COMPUTE command. For information about the COMPUTE command, see [COMPUTE](#) on page 68.

To declare an object in the Procedure Editor using the DECLARE command, use this syntax

```
DECLARE
[ (
  objectname/class;
  .
  .
  .
  ) ]
```

where:

objectname

Is the name of the object that you are creating. The name is subject to the standard naming rules of the Maintain Data language.

class

Is the name of the class of which this object will be an instance.

()

Groups a sequence of declarations into a single DECLARE command. The parentheses are required for groups of local declarations. Otherwise, they are optional.

MNTCON Commands

This topic is a summary of the MNTCON commands that are available in legacy Maintain.

In this appendix:

- [MNTCON CDN_FEXINPUT](#)
 - [MNTCON COMPILE](#)
 - [MNTCON EX](#)
 - [MNTCON EXIT_WARNING](#)
 - [MNTCON MATCH_CASE](#)
 - [MNTCON RADIO_BUTTON_EMIT_TEXT](#)
 - [MNTCON REMOTESTYLE](#)
 - [MNTCON RUN](#)
-

MNTCON CDN_FEXINPUT

By default, you must use a decimal point (.) to indicate a decimal position when writing a value in a Maintain procedure (for example, a COMPUTE statement), and a comma (,) to demarcate thousands, regardless of the CDN setting.

To write the value in a procedure using the format matching the CDN setting for a value other than OFF (for example, ON, QUOTE, QUOTEP, SPACE), use MNTCON CDN_FEXINPUT ON in the EDASPROF file or user profile, and use double quotation marks (") to delimit the value. You can use single quotation marks (') or double quotation marks (") when CDN=ON or SPACE. You must use double quotations marks (") when CDN=QUOTE or QUOTEP.

Example 1:

The following are both correct for all CDN settings by default:

```
COMPUTE MYVAL/D12.2=1234.56;
```

```
COMPUTE MYVAL/D12.2="1,234.56";
```

Example 2:

The following are both correct for SET CDN = QUOTE when the EDASPROF or user profile contains MNTCON CDN_FEXINPUT ON:

```
COMPUTE MYVAL/D12.2="1'234,56";
```

```
COMPUTE MYVAL/D12.2="1234,56";
```

Note: This command does not apply to values entered in a form at run time.

This command is outside the Maintain language, but is described in this content for your convenience.

Syntax: **How to Use the MNTCON CDN_FEXINPUT Command**

When using a CDN value other than OFF, place the following statement in the server profile file (edasprof.prf) or user profile:

```
MNTCON CDN_FEXINPUT {ON|OFF}
```

where:

ON

Allows you to write values in Maintain procedures in the manner used by the actual CDN setting, for example:

- Using a comma (,) to denote a decimal place when CDN=ON, SPACE, or QUOTE.
- Using a single quotation mark (') to demarcate thousands when CDN=QUOTE or QUOTEP.
- Using a space to demarcate thousands when CDN=SPACE.

Follow these rules when writing values using Continental Decimal Notation with MNTCON CDN_FEXINPUT ON:

- You must use single quotation marks (') or double quotation marks (") to delimit values for ON or SPACE.
- You must use double quotation marks (") to delimit values for QUOTE. You must also use double quotation marks (") to delimit values for QUOTEP if you need to write the value with single quotation marks (') to separate thousands.

OFF

Requires you to write values in Maintain procedures using a period (.) to denote a decimal place for all CDN settings. OFF is the default value.

When demarcating thousands, a comma (,) must be used, and the value must be enclosed in quotation marks.

MNTCON COMPILE

The MNTCON COMPILE command creates a compiled Maintain procedure which, under Windows and UNIX, has an extension of .fcm., and under z/OS is allocated to ddname FOCCOMP.

You can reduce the time needed to start a Maintain procedure that contains forms by compiling the procedure. The more frequently the Maintain procedure will be run, the more time you save by compiling it.

This command is outside the Maintain language, but is described in this content for your convenience.

Syntax: How to Use the MNTCON COMPILE Command

The syntax of the MNTCON COMPILE command is

```
MNTCON COMPILE [dirname/]procname
```

where:

dirname

Is the directory name on the Reporting Server where the Maintain procedure is located. This is optional.

procname

Is the name of a Maintain procedure. First, the MNTCON COMPILE command looks for a Maintain procedure with a .mnt extension or a MAINTAIN file type or ddname. If it does not find one, it looks for a Maintain procedure with a .fex extension or a FOCEXEC file type or ddname.

Reference: Commands Related to MNTCON COMPILE

- ❑ **MNTCON RUN.** Executes compiled Maintain procedures.
- ❑ **MNTCON EX.** Executes uncompiled Maintain procedures.

MNTCON EX

You use the MNTCON EX command to run an uncompiled Maintain procedure.

This command is outside the Maintain language, but is described in this content for your convenience.

Syntax: **How to Use the MNTCON EX Command**

To run an uncompiled Maintain procedure (with either a .mnt or .fex extension, or a MAINTAIN or FOCEXEC file type or ddname), use the following syntax

```
MNTCON EX [dirname/procname [-v parm1 , ... parmn]
```

where:

dirname

Is the directory name on the Reporting Server where the Maintain procedure is located. This is optional.

procname

Is the name of a Maintain procedure. First, the MNTCON EX command looks for a Maintain procedure with a .mnt extension or a MAINTAIN file type or ddname. If it does not find one, it looks for a Maintain procedure with a .fex extension or a FOCEXEC file type or ddname.

-v

Is the flag that indicates parameters will be passed to the Maintain procedure. This is optional.

*parm1 ... parm*n**

Can be either positional parameters or parm="value" keyword parameters. Parameter types can be mixed within the same MNTCON EX command line. The maximum number of parameters you can pass is 128. You should separate all parameters using commas (,). You should use single quotation marks (') or double quotation marks (") to enclose values containing spaces or commas (,). Use with Sys_mgr functions (Sys_Mgr.Get_NameParm, Sys.Mgr.Get_InputParams_Count and Sys_Mgr.Get_PositionParm) to retrieve the values. If any of these functions are unsuccessful, FOCERROR is set to -1.

For more information, see [SYS_MGR](#) on page 157.

Reference: **Commands Related to MNTCON EX**

- MNTCON COMPILE.** Compiles Maintain procedures.
- MNTCON RUN.** Executes compiled Maintain procedures.

Invoking Maintain Procedures: Passing Parameters

You can issue MNTCON EX or MNTCON RUN with the flag `-v` to pass input parameters from the command line when invoking Maintain applications, in a manner similar to passing parameters to FOCXECs. This method bypasses the requirement of importing the webbase2 file and coding web client variable retrieval. You can use this syntax in a FOCXEC (.fex) or from within a backend server edastart `-t` session.

Positional and key-matching parameters are supported, and you can use both together in the same Maintain EX or RUN command. Parameters are defined as A0. The maximum number of parameters you can pass is 128. You may include Dialogue Manager commands in a FOCXEC when invoking MNTCON EX or RUN with the `-v` option.

Syntax: How to Use the MNTCON EX Command to Pass Parameters

```
MNTCON [EX|RUN] procname -v "parmlvalue" ... "parmrvalue"
```

where:

procname

Is the name of a Maintain procedure.

parmlvalue ... *parmrvalue*

Can be either positional parameter values in single quotation marks (') or double quotation marks ("), or a `parm="value"` key-matching parameter. You can mix positional and key-matching parameters.

The target Maintain procedure uses Maintain SYS_MGR function subcommands to retrieve the values.

```
sys_mgr.get_positionParm
```

```
sys_mgr.GET_inputparams_count
```

```
sys_mgr.get_nameParm
```

If any of these SYS_MGR functions is not successful, FOCERROR is set to -1.

For more information on SYS_MGR functions, see [SYS_MGR](#) on page 157.

Example: Passing and Retrieving Parameters

```
MNTCON EX START1 -v abc, '24 Houston Center' , ADDR='Cape Canaveral',  
COUNTRY=USA
```

Target Maintain procedure START1 could include:

```
Parm1/a0=sys_mgr.get_positionParm(1);
```

to get the first positional parameter. Here it returns value abc for Parm1.

```
Posvar/i2=sys_mgr.GET_inputparams_count();
```

to return the total number of positional parameters. Here it returns 2 for Posvar.

```
Address/a0=sys_mgr.get_nameParm('ADDR');
```

to return value for key-matching parameter ADDR. Here it returns Cape Canaveral for Address.

Note: Sys_mgr.get_nameParm is case-sensitive. Use the same case for the parameter when retrieving the value as you use when passing it.

For more information on SYS_MGR functions, see [SYS_MGR](#) on page 157.

MNTCON EXIT_WARNING

By default, the exit message, *This application has been disconnected*, appears when a browser session containing an active Maintain application is closed. To control the display of this exit warning, use MNTCON EXIT_WARNING.

This command is outside the Maintain language, but is described in this content for your convenience.

Syntax: How to Use the MNTCON EXIT_WARNING Command

Place the following statement in the server profile file (edasprof.prf) or user profile:

```
MNTCON EXIT_WARNING {ON|OFF}
```

where:

ON

Enables the display of the exit message, *This application has been disconnected*, when an active Maintain browser is closed. ON is the default value.

OFF

Disables the exit warning that displays when an active Maintain browser is closed.

MNTCON MATCH_CASE

By default, segment names and field names in Master Files must be in uppercase. To enable mixed-case names, use MNTCON MATCH_CASE ON.

Syntax: **How to Enable Mixed-Case Naming**

The feature to support mixed-case and NLS characters in the Master File is enabled by the following command in the EDASPROF, user, group, or service profile:

```
MNTCON MATCH_CASE {ON|OFF}
```

where:

OFF

Is the default. Segment names and field names in Master Files still must be uppercase, and Maintain refers to them in mixed-case or lowercase without error.

ON

Means that mixed-case and NLS characters will be respected. Developers must be consistent in their references to named components in terms of the case used.

Components are:

- Case names
- Class names
- Function names
- Object names
- Stack names
- Variable name

With the feature enabled (MNTCON MATCH_CASE ON in a profile):

- Field names COUNTRY, Country, and CounTry all refer to different fields.
- Developers must be consistent with casing when referring to classes, functions, objects, stacks, and variables. For example, CASE MYCASE would need any associated PERFORM statement to refer to the case name as MYCASE.
- IWC. function names must be written exactly as follows:
 - IWC.putCgiData
 - IWC.getCgiData
- MAINTAIN and END are still required to be in uppercase.
- Certain keywords will be automatically translated to uppercase. For example, contains and CONTAINS will always mean the same thing.

Note: With the feature off, there should be no issues running previously developed applications.

With the feature enabled, previously written applications would need to be reviewed, then updated (to keep the case, class, and other names consistent), and finally redeployed.

MNTCON RADIO_BUTTON_EMIT_TEXT

When using a web link event for a radio button control, the selected item is passed as a text value. To send FOCINDEX instead of a text value, use MNTCON RADIO_BUTTON_EMIT_TEXT.

This command is outside the Maintain language, but is described in this chapter for your convenience.

Syntax: **How to Send FOCINDEX From a Radio Button Web Link Event**

Use the following statement in the server profile file (edasprof.prf) or in individual user profiles:

```
MNTCON RADIO_BUTTON_EMIT_TEXT {ON|OFF}
```

where:

ON

Indicates that the text value of the selected item of a radio button will be passed in a web link event. ON is the default value.

OFF

Indicates that FOCINDEX will be used.

MNTCON REMOTESTYLE

By default, Maintain supports the use of a variable for the server name in CALL AT and EXEC AT statements. It is possible to disable the variable server name feature by using MNTCON REMOTESTYLE.

This command is outside the Maintain language, but is described in this content for your convenience.

Syntax: **How to Disable the Variable Server Name Feature**

Use the following statement in the server profile file (edasprof.prf) or in individual user profiles:

```
MNTCON REMOTESTYLE {ON|OFF}
```

where:

ON

Allows variables to be used in the AT SERVER syntax for use with the CALL and EXEC commands. ON is the default value.

OFF

Disables the variable server name feature.

MNTCON RUN

You use the MNTCON RUN command to run a Maintain procedure that has been compiled.

This command is outside the Maintain language, but is described in this content for your convenience.

Syntax: How to Use the MNTCON RUN Command

To run a Maintain procedure that has been compiled (with either a .fcm extension or a FOCCOMP file type or ddname), use the following syntax

```
MNTCON RUN [dirname]/procname [-v parm1 , ... parmn]
```

where:

dirname

Is the directory name on the Reporting Server where the Maintain procedure is located. This is optional.

procname

Is the name of a Maintain procedure.

-v

Is the flag that indicates parameters will be passed to the Maintain procedure. This is optional.

*parm1 ... parm*n**

Can be either positional parameters or parm="value" keyword parameters. Parameter types can be mixed within the same MNTCON RUN command line. The maximum number of parameters you can pass is 128. You should separate all parameters using commas (.). You should use single quotation marks (') or double quotation marks (") to enclose values containing spaces or commas (.). Use with Sys_mgr functions (Sys_Mgr.Get_NameParm, Sys.Mgr.Get_InputParams_Count and Sys_Mgr.Get_PositionParm) to retrieve the values. If any of these functions are unsuccessful, FOCERROR is set to -1.

For more information, see [SYS_MGR](#) on page 157.

Reference: Commands Related to MNTCON RUN

- ❑ **MNTCON COMPILE.** Executes compiled Maintain procedures.
- ❑ **MNTCON EX.** Executes uncompiled Maintain procedures.

For information on passing parameters with MNTCON RUN, see [Invoking Maintain Procedures: Passing Parameters](#) on page 217.

Syntax: How to Use the MNTCON RUN Command to Pass Parameters

```
MNTCON [EX|RUN] procname -v "parmlvalue" ... "parmrvalue"
```

where:

```
parmlvalue ... parmrvalue
```

Can be either positional parameter values in single quotation marks (') or double quotation marks ("), or a parm="value" key-matching parameter. You can mix positional and key-matching parameters.

The target Maintain procedure uses Maintain SYS_MGR function subcommands to retrieve the values.

```
Sys_mgr.get_positionParm
```

```
Sys_mgr.GET_inputparams_count
```

```
Sys_mgr.get_nameParm
```

If any of these SYS_MGR functions is not successful, FOCERROR is set to -1.

For more information on SYS_MGR functions, see [SYS_MGR](#) on page 157.

Example: Passing and Retrieving Parameters

```
MNTCON RUN START1 -v NASA, '24 Houston Center', ADDR='Cape Canaveral',  
COUNTRY=USA
```

Maintain procedure START1 could include:

```
Parm1/a0=sys_mgr.get_positionParm(1);
```

to get the first positional parameter. Here it returns value NASA for Parm1.

```
Parm1/a0=sys_mgr.get_positionParm(3);
```

Here FOCERROR is set to -1, as there were only two positional parameters passed.

```
Posvar/i2=sys_mgr.GET_inputparams_count();
```

to return the total number of positional parameters. Here it returns 2 for Posvar.

```
Address/a0=sys_mgr.get_nameParm('ADDR');
```

to return value for key-matching parameter ADDR. Here it returns Cape Canaveral for Address.

Note: Sys_mgr.get_nameParm is case-sensitive. Use the same case for the parameter when retrieving the value as you use when passing it.

For more information on SYS_MGR functions, see [SYS_MGR](#) on page 157.

Legal and Third-Party Notices

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO, the TIBCO logo, the TIBCO O logo, FOCUS, iWay, Omni-Gen, Omni-HealthData, and WebFOCUS are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle Corporation and/or its affiliates.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the readme file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

This and other products of TIBCO Software Inc. may be covered by registered patents. Please refer to TIBCO's Virtual Patent Marking document (<https://www.tibco.com/patents>) for details.

Copyright © 2021. TIBCO Software Inc. All Rights Reserved.

Index

- subtraction operator [`* a`] [23](#)
- * comment delimiter [18](#)
- * multiplication operator [23](#)
- ** exponentiation operator [23](#)
- *\$ comment delimiter [18](#)
- / division operator [23](#)
- \\ character in character expressions [40](#), [41](#)
- \\n character in character expressions [40](#)
- + operator [23](#)
- | and || characters in character expressions [40](#)
- \$* comment delimiter [18](#)
- \$\$ comment delimiter [18](#)

A

- A0 data type [42](#), [43](#)
- A0 variables [59](#)
 - passing between procedures [59](#), [96](#), [113](#)
- addition in date expressions [33](#)
- addition operator [23](#)
- ALL keyword [68](#)
 - in COMPUTE command [68](#)
 - in COPY command [73](#)
 - in DECLARE command [77](#)
 - in DELETE command [80](#)
 - in INCLUDE command [108](#)
 - in NEXT command [120](#)
 - in REPEAT command [139](#)
 - in REVISE command [147](#)

- ALL keyword [68](#)
 - in UPDATE command [168](#)
- alphanumeric expressions [39](#)
 - backslash (\\) character [40](#), [41](#)
 - concatenating strings [40](#)
 - escape character [40](#)
 - evaluating [40](#)
- alphanumeric format (MODIFY)
 - passing A0 variables to procedures [96](#), [113](#)
- alphanumeric format
 - variable length [42](#), [43](#)
- AND keyword [113](#)
 - in MAINTAIN command [113](#)
 - in NEXT command [120](#)
- AND logical operator [44](#), [45](#)
- applications
 - controlling environment [157](#)
 - debugging [100](#)
- assigning values to variables (Maintain) [53](#)
- AT keyword [59](#), [96](#)
 - in CALL command [59](#)
 - in EXEC command [96](#)
- AUTOCOMMIT command [191](#), [192](#)

B

- backslash (\\) character in character expressions [41](#)
- BEGIN command (Maintain) [58](#)
 - nested BEGIN blocks [59](#)

Boolean expressions [44](#)

broadcast commit [181](#)

C

CALL command [59](#), [115](#)

 MAINTAIN command and [115](#)

canceling commands (Maintain) [17](#)

CASE command [63](#)

 PERFORM command and [65](#)

case sensitivity [12](#)

cases (Maintain) [63](#)

 calling using COMPUTE [72](#)

 return values [66](#)

 Top function [66](#)

cases in functions [200](#)

CDN (Continental Decimal Notation) [28](#)

 setting in a Maintain procedure [160](#)

change verification strategies [191](#)

 for DB2 data sources [191](#), [195](#), [196](#)

 for FOCUS data sources [185](#)

character expressions [39](#)

 \[\](#) character [40](#), [41](#)

 \[\n](#) character [40](#)

 | and || characters [40](#)

 backslash (\[\](#)) character [40](#), [41](#)

 concatenating strings [40](#)

 double quotation marks [40](#), [41](#)

 escape character [40](#)

 evaluating [40](#)

 single quotation marks [40](#), [41](#)

character format [42](#), [43](#)

 variable length [42](#), [43](#)

character strings [40](#), [41](#)

Class Editor [202](#), [207](#)

 Description tab [202](#)

 Functions tab [202](#)

classes [199](#), [200](#)

 defining [199](#), [202–204](#)

 deleting [202](#), [206](#)

 DESCRIBE command and [202](#), [204](#)

 editing [202](#), [206](#)

 inheritance [202](#)

 libraries [118](#), [209](#)

 member functions [200](#)

 member variables [200](#)

 renaming [202](#), [206](#)

 subclasses and superclasses [202–204](#)

CLEAR keyword [154](#)

 in STACK CLEAR command [154](#)

CLOSE keyword in WINFORM command [172](#)

CLOSE_ALL keyword in WINFORM command [172](#)

columns in stacks with virtual fields [72](#)

command types [17](#)

 multi-line [17](#)

commands [17](#)

 canceling [17](#)

comment delimiters [18](#)

 dollar sign asterisk [18](#)

 double dollar sign [18](#)

- comments [18](#)
 - Maintain procedures [16, 18](#)
 - COMMIT command [177](#)
 - DB2 data sources [191, 192](#)
 - defining a logical transaction [177](#)
 - COMMIT parameter [184](#)
 - communication configuration files [190, 191](#)
 - for WebFOCUS Servers [190, 191](#)
 - COMPILE command (Maintain) [215](#)
 - COMPUTE command (Maintain) [68](#)
 - creating user-defined fields [71](#)
 - IF command and [107](#)
 - concurrent processing [182, 185](#)
 - conditional actions (Maintain) [55](#)
 - conditional expressions [46](#)
 - COMPUTE commands [107](#)
 - configuration files for WebFOCUS Servers [190, 191](#)
 - communication configuration files [190, 191](#)
 - EDACSG [190, 191](#)
 - odin.cfg [190, 191](#)
 - CONTAINS logical operator [45](#)
 - in NEXT WHERE phrase [123](#)
 - Continental Decimal Notation (CDN) [28](#)
 - COPY command [73](#)
 - CURRENT keyword in COPY command [73](#)
- D**
- DATA keyword [68](#)
 - in COMPUTE command [68](#)
 - DATA keyword [68](#)
 - in DECLARE command [77](#)
 - data source stacks [13, 111](#)
 - copying rows [73](#)
 - libraries and [119](#)
 - naming [13](#)
 - navigating [126](#)
 - position [146](#)
 - updating [147, 168](#)
 - virtual fields [72](#)
 - data sources
 - command failure [178](#)
 - FOCUS Database Server [185](#)
 - logical transactions [175](#)
 - position within logical transactions [179](#)
 - reading with report procedures [186](#)
 - sharing [182](#)
 - data types [200](#)
 - classes [199, 202–204](#)
 - matching in function parameters [65](#)
 - date and time expressions [28](#)
 - date expressions [28](#)
 - addition and subtraction in [33](#)
 - components [31](#)
 - constants in [31](#)
 - evaluating [29](#)
 - extracting [31](#)
 - formats [29, 30](#)
 - manipulating in date format [31](#)
 - operand format [32](#)

date-time data types [34](#), [36](#)

 assigning [36](#)

 comparing [36](#)

 describing [34](#)

 functions [37](#)

 ISO standards [39](#)

 missing values and [36](#)

date-time formats [34](#)

date-time values [34](#)

 assigning [34](#)

DATEDISPLAY parameter [160](#)

DB2 data sources [191](#)

 change verification [191](#), [195](#), [196](#)

 data adapter differences [191](#), [192](#)

 transaction processing [191](#), [193](#), [195](#), [196](#)

DBMS_ERRORCODE command [157](#), [158](#)

DECLARE command [77](#), [210](#), [211](#)

DEFCENT parameter [160](#)

DEFINE attribute

 in Master File [72](#)

DELETE command [80](#)

 deleting data [80](#)

deploying applications [190](#)

DESCRIBE command [202](#), [204](#)

Description tab

 in Class Editor [202](#)

DFC keyword [68](#)

 in COMPUTE command [68](#)

 in DECLARE command [77](#)

DFC parameter [160](#)

directory paths [40](#), [41](#)

DIV operator [23](#), [24](#)

division operator [23](#)

dollar sign asterisk comment delimiter [18](#)

double dollar sign comment delimiter [18](#)

DROP keyword [59](#), [96](#)

 in CALL command [59](#)

 in EXEC command [96](#)

DROP TABLE command from Maintain [159](#)

duplicate names [13](#)

E

EDACSG ddname [190](#), [191](#)

EDASPROF global server profile [188](#)

 FOCUS Database Server [188](#)

Edit Class dialog box [202](#), [207](#)

ELSE keyword [46](#), [104](#), [106](#)

 in conditional expressions [46](#)

 in IF command [104](#)

EMGSRV parameter [160](#)

END command [96](#)

END keyword in GOTO command [179](#)

END keyword

 in GOTO command [101](#)

ENDBEGIN keyword [58](#)

ENDCASE [63](#), [101](#)

 command [63](#)

 keyword in GOTO command [101](#)

ENDREPEAT [101](#), [139](#)

 command [139](#)

- ENDREPEAT [101, 139](#)
 - keyword in GOTO command [101](#)
 - ENGINE command [158](#)
 - environment variables [190, 191](#)
 - HOME [190, 191](#)
 - EQ logical operator [45](#)
 - in NEXT WHERE phrase [123](#)
 - EQ_MASK logical operator in NEXT WHERE phrase [123](#)
 - error messages [100, 157, 158](#)
 - displaying [100](#)
 - retrieving from DBMSs [157, 158](#)
 - escape characters [40, 41](#)
 - in character expressions [40](#)
 - EX command [215](#)
 - EXCEEDS logical operator [123](#)
 - in NEXT WHERE phrase [123](#)
 - EXEC command [96](#)
 - EXIT keyword in GOTO command [101](#)
 - EXIT_WARNING command (Maintain) [218](#)
 - EXITREPEAT keyword in GOTO command [101](#)
 - exponentiation operator [23](#)
 - expressions [21](#)
 - Boolean [44](#)
 - character [39](#)
 - conditional [46, 107](#)
 - date and time [28](#)
 - default values [47](#)
 - limits in Maintain [21–23](#)
 - logical [44](#)
 - expressions [21](#)
 - null values [47](#)
 - numeric [23](#)
 - relational [44](#)
 - external procedures [190](#)
 - deployment considerations [190](#)
 - reporting [190](#)
 - extracting substrings [40](#)
- F**
- FALSE value for logical expressions [44, 45](#)
 - FDS (FOCUS Database Server) [184](#)
 - change verification [185](#)
 - deployment considerations [190](#)
 - identifying [188](#)
 - report procedures [189, 190](#)
 - SET PATHCHECK [187](#)
 - transaction processing [186](#)
 - FETCH SQL command equivalent (Maintain) [119](#)
 - fields (Maintain)
 - naming [13](#)
 - null values [47, 173](#)
 - FILE keyword
 - in MAINTAIN command [113](#)
 - FILES keyword in MAINTAIN command [113](#)
 - flow of control (Maintain) [59](#)
 - CALL command [59](#)
 - CASE command [63](#)
 - GOTO command [101](#)
 - IF command [104](#)

- flow of control (Maintain) [59](#)
 - looping [139](#)
 - ON MATCH command [134](#)
 - ON NEXT command [135](#)
 - ON NOMATCH command [135](#)
 - ON NONEXT command [136](#)
 - PERFORM command (Maintain) [137](#)
 - REPEAT command [139](#)
- FOCCOMP ddname/file type [215](#)
- FocCount variable [98](#)
- FocCurrent variable [98](#), [182](#)
 - change verify protocol [185](#)
 - with COMMIT command [67](#)
- FocError variable [98](#)
- FocErrorRow variable [99](#)
- FocFetch variable [99](#)
- FocIndex variable [99](#)
- FocMsg [100](#)
- FocMsg stack [100](#)
- FOCSET command [159](#), [160](#)
- FOCUS data sources [184](#), [188](#)
 - change verification [186](#)
 - concurrent transactions [184](#)
 - FOCUS Database Server [185](#)
 - SET COMMIT [184](#), [185](#)
 - SET PATHCHECK parameter [187](#)
 - sharing access [185](#)
 - transaction processing [184](#), [187](#)
- FOCUS Database Server (FDS) [184](#)
 - change verification [186](#)
 - FOCUS Database Server (FDS) [184](#)
 - deployment considerations [190](#)
 - identifying [188](#)
 - report procedures [189](#)
 - SET PATHCHECK [187](#)
 - transaction processing [185](#)
- FOR keyword [73](#)
 - in COPY command [73](#)
 - in DELETE command [80](#)
 - in INCLUDE command [108](#)
 - in NEXT command [120](#), [123](#)
 - in REVISE command [147](#)
 - in UPDATE command [168](#)
- forms (Maintain)
 - displaying at run time [172](#)
 - displaying default values [173](#)
 - libraries and [119](#)
- FROM keyword [59](#)
 - in CALL command [59](#)
 - in COPY command [73](#)
 - in DELETE command [80](#)
 - in EXEC command [96](#)
 - in INCLUDE command [108](#)
 - in MAINTAIN command [113](#)
 - in MATCH command [116](#)
 - in REVISE command [147](#)
 - in UPDATE command [168](#)
- functions [200](#)
 - member [200](#)

G

GE logical operator [45](#)

General tab

Class Editor [202](#)

GET keyword in WINFORM command [172](#)

GET_PREMATCH command [164](#)

global variables [80](#)

GOTO command (Maintain) [101](#)

data source commands and [103](#)

ENDCASE command and [104](#)

PERFORM command and [104](#), [139](#)

GOTO END command [179](#), [180](#)

GT logical operator [45](#)

H

HIDE keyword in WINFORM command [172](#)

HIGHEST keyword in STACK SORT command [155](#)

HOME environment variable [190](#), [191](#)

I

IF command (Maintain) [104](#)

keyword in conditional expressions [46](#)

IMPORT keyword in MODULE command [118](#), [209](#)

import modules

using as class libraries [209](#)

importing modules [118](#)

restrictions [119](#)

IN logical operator in NEXT WHERE phrase [123](#)

INCLUDE command [107](#), [108](#)

adding data [107](#)

INCLUDE command [107](#), [108](#)

data source position [110](#)

null values [111](#)

unique segments [108](#)

INFER command [111](#), [112](#)

creating stacks [111](#)

inheritance [202](#)

INTO keyword [59](#)

in CALL command [59](#)

in COPY command [73](#)

in EXEC command [96](#)

in INFER command [112](#)

in MAINTAIN command [113](#)

in MATCH command [116](#)

in NEXT command [120](#)

IS logical operator in NEXT WHERE phrase [123](#)

IS_NOT logical operator in NEXT WHERE phrase [123](#)

issuing DROP TABLE command [159](#)

K

KEEP keyword [101](#), [179](#)

in CALL command [59](#)

in EXEC command [96](#)

in GOTO command [101](#)

L

LANGUAGE parameter [160](#)

Language Wizard [51](#)

LE logical operator [45](#)

libraries (Maintain)

importing [118](#)

restrictions [119](#)

line feeds in character expressions [40](#)

local variables [80](#)

log files (Maintain) [151](#)

SAY command [151](#)

segment and stack values [152](#)

TYPE command [165](#)

logical expressions [44](#)

Boolean expressions [44](#)

evaluating [45](#)

operators [45](#)

relational expressions [44](#)

logical operators [45](#)

logical transactions [175](#), [182](#)

broadcast commit [181](#)

concurrent transactions [182](#), [183](#)

concurrent transactions in FOCUS Database
Server [185](#)

data source position [179](#)

DBMS types [181](#)

defining [177](#)

deployment considerations [190](#)

ending an application [181](#)

failure [178](#)

FocCurrent [182](#)

multiple data source types [181](#)

multiple servers [181](#)

logical transactions [175](#), [182](#)

open transactions when an application ends
[181](#)

processing [175](#), [176](#)

rolling back [178](#)

spanning procedures [179](#)

success [181](#)

looping in Maintain language [53](#)

loops [141](#), [142](#)

branching out of [145](#)

ending [145](#)

simple [141](#)

LT logical operator [45](#)

M

MAINTAIN command [113](#), [114](#)

calling a procedure from another procedure
[115](#)

specifying data sources [115](#)

Maintain Data libraries [209](#)

Maintain Data procedures [179](#), [188](#)

MODIFY [191](#)

sharing data sources with App Studio

Maintain Data [191](#)

transaction integrity [179](#)

Maintain functions [51](#)

calling using COMPUTE [72](#)

calling using PERFORM [137](#), [138](#)

defining [63](#)

passing parameters [65](#)

- Maintain functions [51](#)
 - return values [66](#)
 - Top [66](#)
 - Maintain language [11, 51](#)
 - case sensitivity [12](#)
 - class libraries [57](#)
 - commands [17, 51–53](#)
 - comments [16, 18](#)
 - conditional actions [55](#)
 - displaying forms [53](#)
 - function libraries [57](#)
 - loops [53](#)
 - manipulating stacks [53](#)
 - messages and logs [57](#)
 - multi-line commands [17](#)
 - naming rules [12, 13, 15](#)
 - reading data [54](#)
 - transferring control [52](#)
 - variables [53](#)
 - writing transactions [56](#)
 - Maintain procedures [16, 18](#)
 - blank lines [16](#)
 - calling [96](#)
 - comments [18](#)
 - compiling [215](#)
 - running [52](#)
 - MATCH command [116–118](#)
 - NEXT command and [126](#)
 - REPOSITION command and [146](#)
 - MATCH keyword in ON MATCH command [134](#)
 - member functions [200](#)
 - inheritance [202](#)
 - member variables [200, 201](#)
 - inheritance [202](#)
 - MESSAGE parameter [160](#)
 - MISSING attribute [47, 173](#)
 - date-time data type and [36](#)
 - MISSING constant [47, 48](#)
 - missing data and INCLUDE command [111](#)
 - MISSING keyword [68](#)
 - in COMPUTE command (Maintain) [68](#)
 - in DECLARE command (Maintain) [77](#)
 - MNTCON COMPILE command [215](#)
 - MNTCON EX command [215](#)
 - MNTCON EXIT_WARNING command [218](#)
 - MNTCON RADIO_BUTTON_EMIT_TEXT command [220](#)
 - MNTCON REMOTESTYLE command [220](#)
 - MNTCON RUN command [221](#)
 - MOD operator [24](#)
 - MODIFY applications [191](#)
 - modular processing [59](#)
 - MODULE command [209](#)
 - MODULE IMPORT command [118](#)
 - modules [119](#)
 - importing [118](#)
 - multiplication operator [23](#)
- N**
- names [13](#)

naming conventions [15](#)

 reserved words [15](#)

native-mode arithmetic [25](#)

NE logical operator [45](#)

 in NEXT WHERE phrase [123](#)

NE_MASK logical operator in NEXT WHERE phrase
[123](#)

NEEDS keyword [68, 77](#)

 in COMPUTE command [68](#)

 in DECLARE command [77](#)

nested BEGIN blocks [58, 59](#)

nesting IF commands [106](#)

New Class dialog box [202, 207](#)

NEXT command (Maintain) [119–121](#)

 with MATCH command [126](#)

 copying data [122](#)

 data source navigation [126, 128, 129, 131](#)

 loading multi-path transaction data [122](#)

 reading data [126](#)

 REPOSITION command [146](#)

 retrieving rows [124](#)

 unique segments [133](#)

 with multiple rows [123](#)

 with path instances [129](#)

NEXT keyword in ON NEXT command [135](#)

NODATA parameter [173](#)

 null representation [47](#)

 setting in a Maintain procedure [160](#)

NOMATCH keyword in ON NOMATCH command
[136](#)

NONEXT keyword in ON NONEXT command [137](#)

NOT logical operator [44, 45](#)

NOT_IN logical operator in NEXT WHERE phrase
[123](#)

null values [47, 48, 68, 111, 173](#)

 COMPUTE command [68](#)

 expressions [47](#)

 forms [173](#)

 INCLUDE command [111](#)

 testing [48](#)

numeric expressions [23](#)

 Continental Decimal Notation (CDN) [28](#)

 DIV [23, 24](#)

 evaluating [25](#)

 identical operand formats [26](#)

 MOD [24](#)

 operand formats [26, 27](#)

 operators [23](#)

 truncating decimal values [27](#)

O

objects [199, 210](#)

 creating [53](#)

 declaring [77, 210, 211](#)

 Variable Editor [210](#)

odin.cfg file [190, 191](#)

OFF keyword [68, 77](#)

 in COMPUTE command [68](#)

 in DECLARE command [77](#)

OMITS logical operator [45](#)
 in NEXT WHERE phrase [123](#)
 ON keyword [68](#), [77](#), [113](#), [165](#)
 in COMPUTE command [68](#)
 in DECLARE command [77](#)
 in MAINTAIN command [113](#)
 in TYPE command [165](#)
 ON MATCH command [134](#)
 ON NEXT command [135](#)
 ON NOMATCH command [135](#), [136](#)
 ON NONEXT command [136](#), [137](#)
 ON TABLE SET command [153](#)
 operand format for dates [32](#)
 OR logical operator [44](#), [45](#)

P

parameters [65](#)
 PASS setting [160](#), [161](#)
 PATHCHECK parameter [187](#)
 PERFORM command (Maintain) [65](#), [137](#)
 data source commands and [139](#)
 GOTO command and [104](#), [139](#)
 nesting [139](#)
 performance transaction processing [179](#)
 PRE_MATCH setting [163](#)
 presentation logic
 WINFORM command [172](#)
 procedures (Maintain) [18](#)
 blank lines [16](#)
 calling [96](#)

procedures (Maintain) [18](#)
 comments [18](#)
 compiling [215](#)

R

RADIO_BUTTON_EMIT_TEXT command (Maintain)
[220](#)
 records
 selecting [116](#), [119](#)
 REFRESH keyword in WINFORM command [172](#)
 relational expressions [44](#)
 REMOTESTYLE command (Maintain) [220](#)
 REPEAT command (Maintain) [139](#), [142](#), [145](#)
 branching out of loops [145](#)
 ending loops [145](#)
 establishing counters [143](#)
 nested loops [143](#), [144](#)
 simple loops [141](#)
 UNTIL [142](#)
 WHILE [142](#)
 reports
 reading transaction data [186](#)
 REPOSITION command (Maintain) [146](#)
 RESET keyword
 in GOTO command [101](#)
 in WINFORM command [172](#)
 return values for functions [66](#)
 RETURNS keyword in CASE command [63](#)
 REVISE command [147](#), [149](#)

ROLLBACK command [150](#), [177](#), [178](#)

 DB2 data sources [191](#), [192](#)

rounding numeric values [27](#)

RUN command [221](#)

S

SAY command [151](#), [152](#)

 choosing between SAY and TYPE commands
 [152](#)

SEG prefix with SAY command [152](#)

segments [133](#)

 DELETE command [83](#)

 INCLUDE command [108](#)

 NEXT command [133](#)

 REVISE command [147](#)

SELECT SQL command equivalent (Maintain) [119](#)

SET command [152](#), [153](#)

SET keyword in WINFORM command [172](#)

SET parameters [153](#), [184](#)

 COMMIT in FOCUS data sources [184](#), [185](#)

 NODATA [47](#), [173](#)

 PATHCHECK [187](#)

 PREMATCH [164](#)

 setting in Maintain procedure with

 SYS_MGR.FOCSET [160](#)

 with ON TABLE [153](#)

SET PATHCHECK parameter [187](#)

SET_PREMATCH setting [164](#)

SHOW keyword in WINFORM command [172](#)

 SHOW_ACTIVE keyword in WINFORM command
 [172](#)

 SHOW_AND_EXIT keyword in WINFORM command
 [172](#)

 SHOW_INACTIVE keyword in WINFORM command
 [172](#)

SOME keyword [68](#), [77](#)

 in COMPUTE command [68](#)

 in DECLARE command [77](#)

SORT keyword in STACK SORT command [155](#)

STACK CLEAR command [154](#), [155](#)

stack commands

 creating [111](#)

 STACK CLEAR [154](#)

 STACK SORT [155](#)

STACK keyword in COPY command [73](#)

STACK prefix with SAY command [152](#)

STACK SORT command [155](#), [156](#)

stack variables [53](#)

 FocCount [98](#)

 FocIndex [99](#)

stacks

 copying stack rows [73](#)

 creating stacks with INFER [111](#)

 clearing [154](#)

 creating [111](#)

 editing [53](#), [73](#)

 naming [13](#)

 non-data source columns [113](#)

 rows [155](#)

- stacks
 - STACK SORT command [155](#)
 - virtual fields [72](#)
 - strong concatenation [40](#)
 - substrings
 - extracting [41](#)
 - subtraction in date expressions [33](#)
 - subtraction operator [23](#)
 - SYS_MGR global object [157](#)
 - SYS_MGR.DBMS_ERRORCODE command [157](#), [158](#)
 - SYS_MGR.ENGINE command [158](#)
 - SYS_MGR.FOCSET command [159](#), [160](#)
 - SYS_MGR.GET_PREMATCH command [164](#)
 - SYS_MGR.PRE_MATCH command [163](#)
 - SYS_MGR.SET_PREMATCH command [164](#)
 - system variables (Maintain) [53](#), [54](#), [56](#), [99](#)
 - FocCurrent [98](#)
 - FocError [98](#)
 - FocErrorRow [99](#)
 - FocFetch [99](#)
- T**
- TAKES keyword in CASE command [63](#)
 - temporary fields and columns (Maintain)
 - in data sources [72](#)
 - text expressions [39](#)
 - text format [96](#)
 - passing variables to procedures [96](#)
 - variable length [42](#), [43](#)
 - THEN keyword [46](#), [104](#)
 - in conditional expressions [46](#)
 - in IF command [104](#)
 - time expressions [28](#)
 - TO keyword in SAY command [151](#)
 - Top function [66](#), [209](#)
 - libraries and [119](#)
 - TRACEOFF setting [160](#)
 - TRACEON setting [160](#)
 - TRACEUSER setting [160](#)
 - transaction data sources [122](#)
 - transaction integrity [175–177](#), [182](#), [191](#)
 - across procedures [179](#)
 - change verification [186](#)
 - concurrent transactions [182](#)
 - deployment considerations [190](#)
 - FocCurrent [182](#)
 - multiple servers [181](#)
 - transaction locking strategies [191](#), [193](#), [195](#)
 - transaction processing [175](#), [182](#)
 - broadcast commit [181](#)
 - change verification [186](#)
 - collecting values [172](#)
 - COMMIT command [67](#)
 - concurrent transactions [185](#)
 - DELETE command [80](#)
 - INCLUDE command [107](#)
 - multiple data source types [181](#)
 - multiple DBMSs rollback [151](#)
 - performance [179](#)

transaction processing [175, 182](#)

- reading multiple-path data sources [122](#)
- REVISE command [147](#)
- ROLLBACK command [150, 151](#)
- UPDATE command [168](#)
- USE command [188](#)

transaction variables (Maintain) [54, 56](#)

- FocCurrent [98](#)
- FocError [98](#)
- FocErrorRow [99](#)
- FocFetch [99](#)
- UPDATE command [171](#)

troubleshooting [100](#)

- displaying WebFOCUS Server messages [100](#)

TRUE value for logical expressions [44, 45](#)

TX data type [42, 43](#)

TYPE (Maintain) [166, 167](#)

TYPE command [165](#)

- choosing between SAY and TYPE commands [152](#)
- embedding spacing information [166](#)
- including variables in a message [166](#)
- justifying variables [167](#)
- multi-line message strings [167](#)
- truncating [167](#)
- writing information to a file [168](#)

U

UNHIDE keyword in WINFORM command [172](#)

unique segments (Maintain) [133](#)

- DELETE command [83](#)
- INCLUDE command [108](#)
- NEXT command [133](#)
- UPDATE [172](#)
- UPDATE command [172](#)

UNTIL keyword in REPEAT command [139](#)

UPDATE command (Maintain) [168–170](#)

- data source position [171](#)
- stacks and [171](#)
- transaction variables [171](#)
- unique segments [172](#)

updating data sources [147](#)

- REVISE command [147](#)
- UPDATE command [168](#)

USE command with FOCUS Database Server [188](#)

USER setting [160](#)

V

variable-length character format [59, 96](#)

- passing A0 variables to procedures [113](#)
- passing variables between procedures [59, 96](#)

variable-length strings [42, 43](#)

variables [68, 200, 210](#)

- creating [53](#)
- assigning values with COMPUTE [68](#)
- declaring [68, 77](#)
- defining objects [210, 211](#)
- local vs. global [80](#)
- system [53, 54, 56](#)

virtual fields in data sources [72](#)

W

WARNING setting [160](#)

weak concatenation [40](#)

WebFOCUS Servers

- changing environment settings [152](#), [159](#)

- communication configuration files [190](#), [191](#)

- configuration directory [190](#), [191](#)

- displaying messages [100](#)

- EDACSG [190](#), [191](#)

- edaserve [190](#), [191](#)

- FOCUS Database Server profiles [188](#)

- logical transactions spanning servers [181](#)

WebFOCUS Servers

- odin.cfg [190](#), [191](#)

WHERE keyword [73](#), [123](#)

- in COPY command [73](#)

- in NEXT command [120](#), [123](#)

WHILE keyword in REPEAT command [139](#)

WINFORM command [172](#)

wizards [51](#)

Y

YRT keyword [68](#)

- in COMPUTE command [68](#)

- in DECLARE command [77](#)

YRTHRESH setting [160](#)

