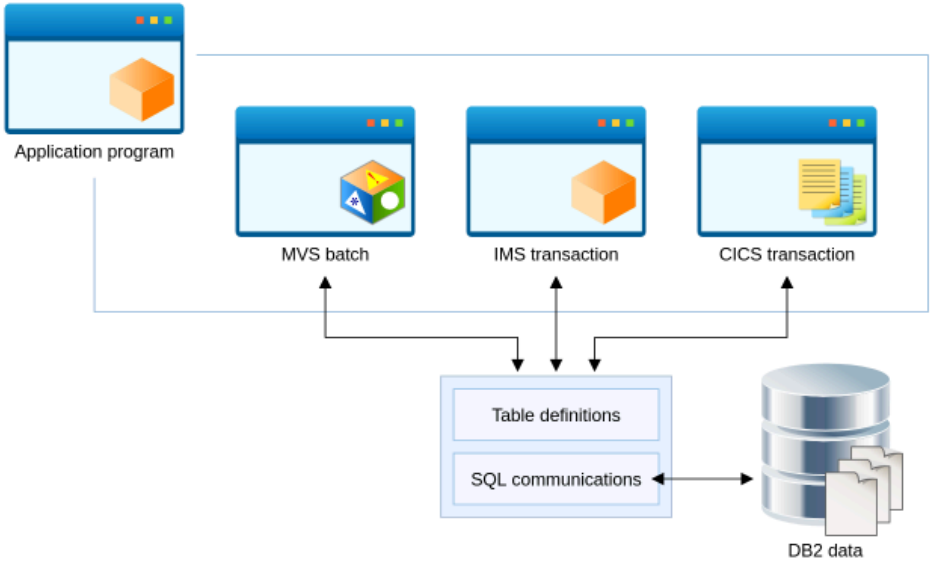
Manipulating Data and Components of an Application Program

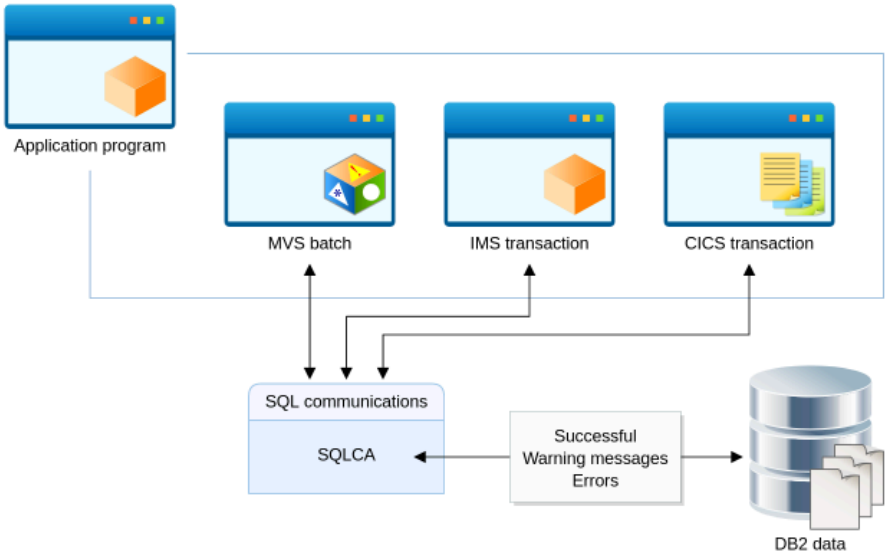
In the previous module you saw how the application program and its embedded SQL statements exist in the source code. In this module you will take a closer look at some of the specific SQL code that you will need to be able to pass values between Db2 and the application program, and to analyze whether the SQL execution has been successful.



It is likely that you will need to add some additional SQL statements to your Db2 program to assist it with its processing, and to provide feedback on its success. This is possible through:

- The declaration of tables and views
- Use of the SQL Communications Area (SQLCA) to view the status of executed SQL statements

We will look first at the SQLCA structure and the valuable information it stores, and can provide to you.



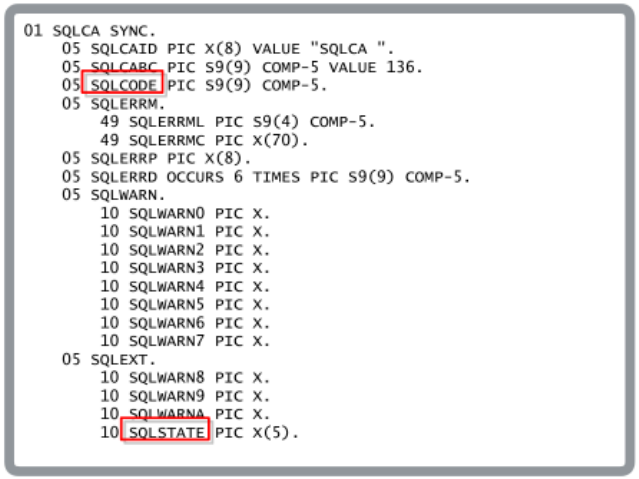
The SQLCA is a structure that is used to receive and store codes and flags returned by Db2, which indicate the success or failure of an SQL statement. The SQLCA has a set structure and can be generated by Db2 for each language.

You can use this generated structure or provide your own specified standalone variables. It is recommended you use the standard structure because it is better understood and contains extra information that can be very useful.

Examples of SQLCA declarations

COBOL	PL/I
EXEC SQL INCLUDE SQLCA END-EXEC.	EXEC SQL INCLUDE SQLCA;

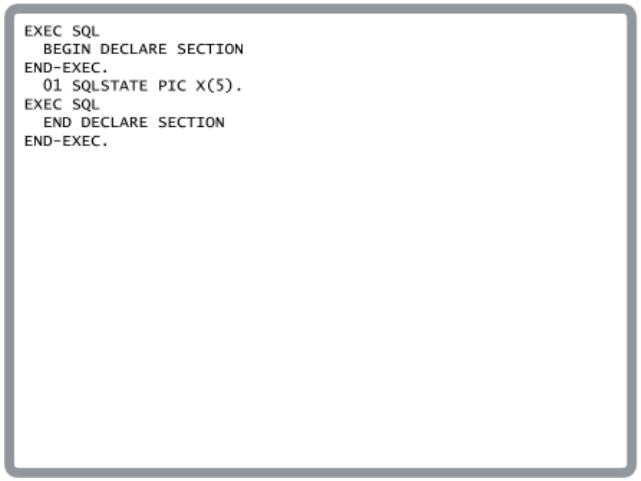
If the SQLCA is to be used by the application program then it needs to be declared. In all host languages except REXX, the SQL INCLUDE statement can be used to provide this information.



```
01 SQLCA SYNC.
    05 SQLCAID PIC X(8) VALUE "SQLCA ".
    05 SQLCABC PIC S9(9) COMP-5 VALUE 136.
    05 SQLCODE PIC S9(9) COMP-5.
    05 SQLERRM.
        49 SQLERRML PIC S9(4) COMP-5.
        49 SQLERRMC PIC X(70).
    05 SQLERRP PIC X(8).
    05 SQLERRD OCCURS 6 TIMES PIC S9(9) COMP-5.
    05 SQLWARN.
        10 SQLWARN0 PIC X.
        10 SQLWARN1 PIC X.
        10 SQLWARN2 PIC X.
        10 SQLWARN3 PIC X.
        10 SQLWARN4 PIC X.
        10 SQLWARN5 PIC X.
        10 SQLWARN6 PIC X.
        10 SQLWARN7 PIC X.
    05 SQLEXT.
        10 SQLWARN8 PIC X.
        10 SQLWARN9 PIC X.
        10 SQLWARNA PIC X.
        10 SQLSTATE PIC X(5).
```

The example here is a description of the SQLCA that is provided when INCLUDE SQLCA is specified for a COBOL program. The following two host variables are discussed in more detail on the pages that follow:

- SQLCODE This is a four byte integer field that contains the Db2 return code.
- SQLSTATE This is a five character string that is the ANSI/ISO standard SQL return code.

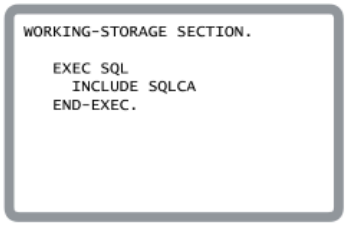


```
EXEC SQL
  BEGIN DECLARE SECTION
END-EXEC.
  01 SQLSTATE PIC X(5).
EXEC SQL
  END DECLARE SECTION
END-EXEC.
```

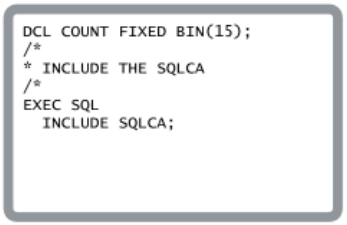
The differences between SQLSTATE and SQLCODE are:

- SQLSTATE is the ANSI/ISO standard SQL return code.
- SQLCODE is the Db2 return code.
- If the program is prepared with the STDSQL(YES) SQL processing option, then you will need to declare host variables for either SQLCODE or SQLSTATE, or both, and the variables must be declared in the DECLARE SECTION of the program.

The graphic here shows an example in COBOL.



COBOL



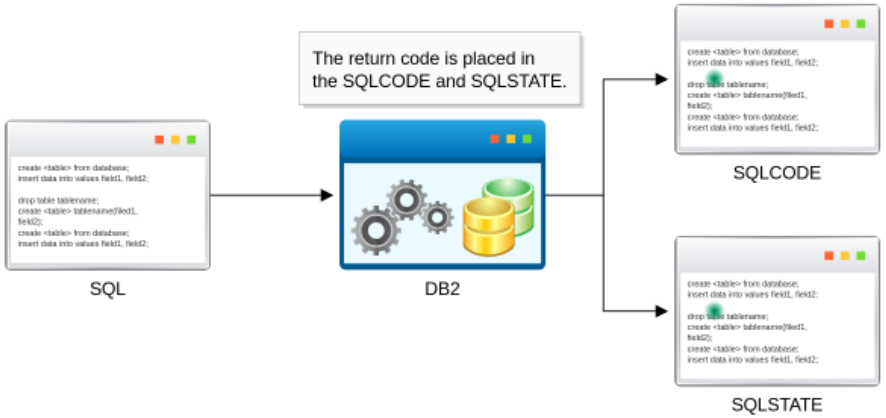
PL/I

In a COBOL program the SQLCA is placed in the WORKING STORAGE or the LINKAGE SECTION. In a PL/I program it is usually placed at the top of the program with the other declares.

Notice the EXEC SQL statement that precedes the SQL. All SQL statements are placed between EXEC SQL and the terminator (END-EXEC or a ;).

The precompile step removes these statements and substitutes host language statements.

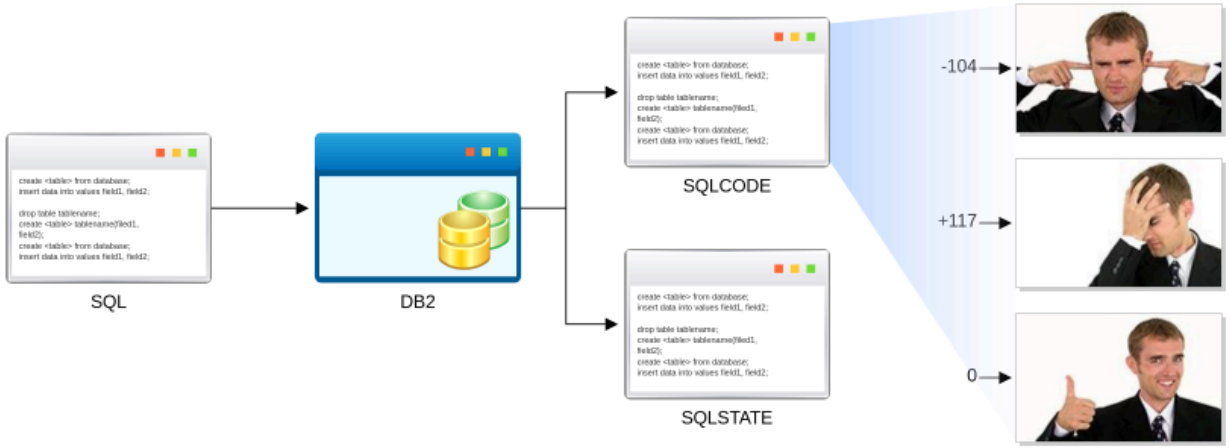
Note that the period after END-EXEC is optional, and in some scenarios will cause problems if used. For example, if the END-EXEC. statement appears within an IF…THEN set of statements, the ending period will inadvertently end the IF statement.



When an SQL statement is executed, the return code is placed in the SQLCODE and the SQLSTATE.

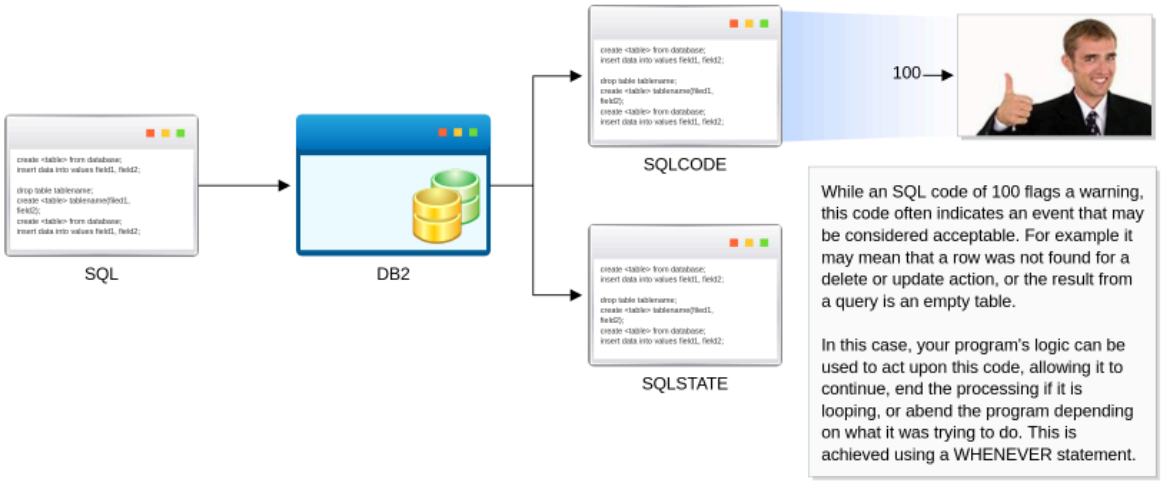
It is usual for programmers to check the return code in SQLCODE or SQLSTATE and take appropriate action.

Click **Play** to see this process.



The values returned in the SQLCODE can be split into three groups. These are:

- Severe errors that are bad enough to terminate the program; these have negative values
- Warnings that have a positive value
- Success that has a zero value



SQLCODE = -811 means that a severe error has occurred; therefore, terminate the program

SQLCODE = 0 means success; therefore, continue processing

SQLCODE = 100 states that no row or no more rows have been found

In most applications if the procedure is SQLCODE = 0, you can continue processing.

```
EXEC SQL
 WHENEVER SQLERROR
 GO TO Z999-ABORT
END-EXEC.
```

```
EXEC SQL
 WHENEVER SQLERROR
 GO TO DB2ERROR
;
```

The only two things that can be done in a WHENEVER statement are branch to a label or CONTINUE.

After branching to a label any action required can be taken, such as writing out the SQLCA to a file and then abending the program. Most sites have some standard code that is included at this point in the program. Db2 supplies a standard module that can be used to format the SQLCA called DSNTIAR (DSNTIAC for CICS programs).

Using a WHENEVER statement, it is possible to automatically branch to an error routine when a severe error is encountered.

Example code trapping a +100 warning:

```
EXEC SQL
 WHENEVER NOT FOUND
 CONTINUE
END-EXEC.
```

One other condition that can be trapped is NOT FOUND which is the equivalent of an SQLCODE of +100. It is therefore a warning, but it means "There are no rows that satisfy your search condition".

This return code is important and the code shown here is possible.

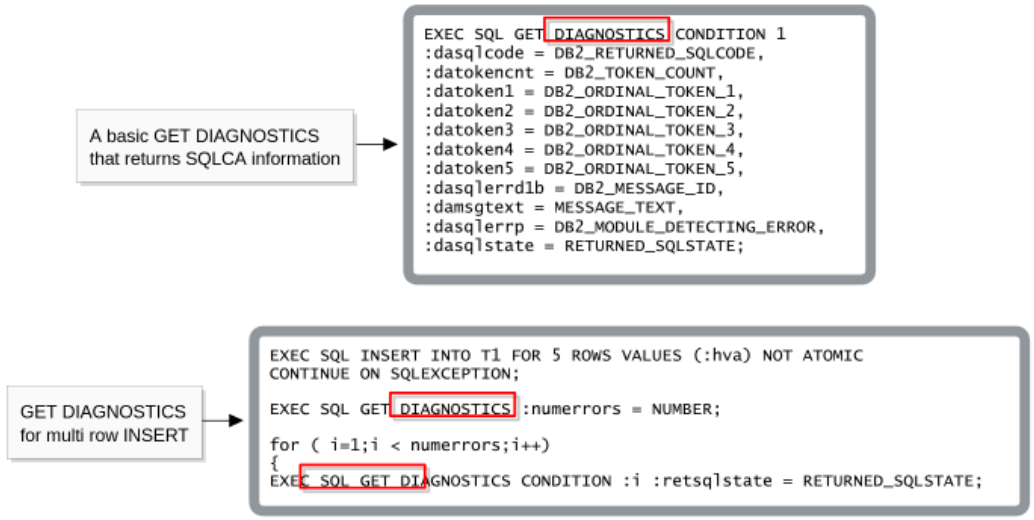
Example code trapping a warning:

```
EXEC SQL
WHENEVER SQLWARNING GO TO ENDDATA
END EXEC.
```

```
EXEC SQL
WHENEVER SQLWARNING CONTINUE
END EXEC.
```

```
EXEC SQL
WHENEVER SQLWARNING GOTO WARNING_HANDLER
END EXEC.
```

It is also possible to trap the more general SQLWARNING. This is any positive SQLCODE, including NOT FOUND.



The GET DIAGNOSTICS statement can also be used to assist in analyzing errors. This statement checks the last SQL statement processed and returns diagnostic data about:

- the SQL statement as a whole
- each error or warning that occurred during the execution of the statement

The resulting information is assigned to host variables that can be further interrogated by the program.

These are example declare statements in a COBOL program.

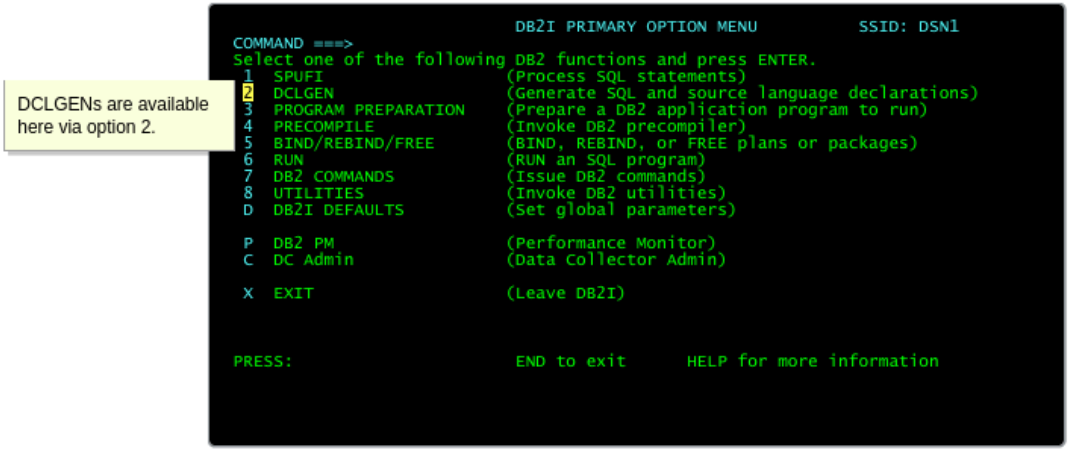
```
EXEC SQL
 DECLARE DB2201.EMPLOYEE TABLE
   (EMP_ID    CHAR(6)           NOT NULL,
    NAME      VARCHAR(20)       NOT NULL,
    SALARY    DECIMAL(7, 2)     NOT NULL,
    AGE       SMALLINT )
END-EXEC.
* COBOL DECLARATION FOR TABLE EMPLOYEE
01 DCLEMPLOYEE.
   10 EMP_ID    PIC X(6).
   10 NAME      PIC X(20).
   10 SALARY    PIC S9(5)V9(2) USAGE COMP-3.
   10 AGE       PIC S9(4) COMP.
```

In this example, DCLGEN has been used to generate a table declaration and corresponding COBOL record description for the table DB2201.EMPLOYEE table. The declarations were stored in the data set member DECEMP.

```
EXEC SQL
  INCLUDE DECEMP
END-EXEC.
```

Before your program can invoke SQL statements such as select, insert, update, or delete, the tables and views that those statements access need to be declared by the program. This provides the precompiler with information used to check your embedded SQL statements.

Declare statements can be entered manually by the user, or in the case of C, COBOL, and PL/I programs, they can be generated using the Db2 declarations generator (DCLGEN).



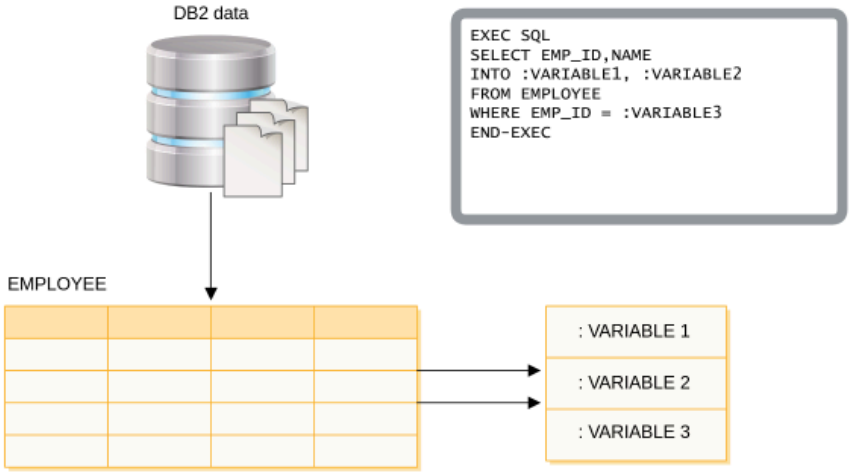
DCLGENs are generated by Db2 on request, usually by the DBAs. Alternatively, they are generated via the DB2I option if you have access, or even in batch.

The DCLGEN:

- Are used by the precompiler to check that the columns match the fields used in the application program
- Can provide the programmer with a host variable for every column used in the program
- Are not compulsory; you are free to use any variables you like, provided the data types match
- Automates the declaration process, simplifying the programmer's work and making it less prone to human error

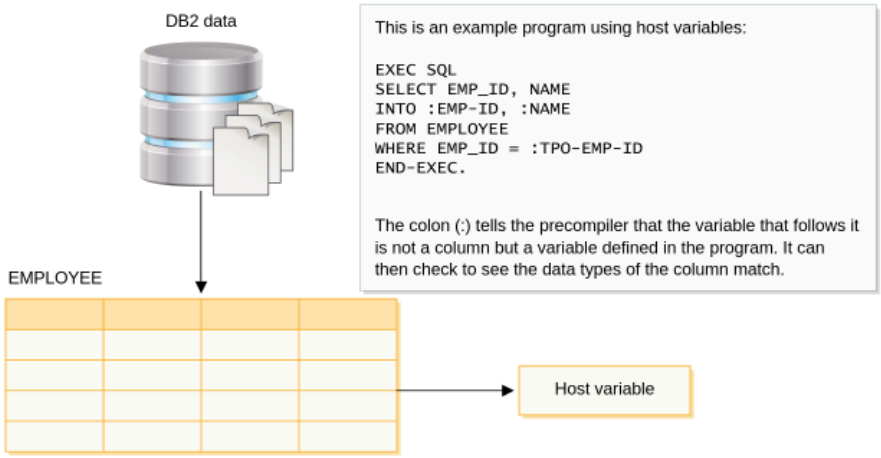
```
EXEC SQL
 DECLARE DB2201.EMPLOYEE TABLE
   (EMP_ID     CHAR(6)           NOT NULL,
    NAME       VARCHAR(20)       NOT NULL,
    SALARY     DECIMAL(7, 2)     NOT NULL,
    AGE        SMALLINT )
END-EXEC.
* COBOL DECLARATION FOR TABLE EMPLOYEE
01 DCLEMPLOYEE.
   10 EMP_ID    PIC X(6).
   10 NAME      PIC X(20).
   10 SALARY    PIC S9(5)V9(2) USAGE COMP-3.
   10 AGE       PIC S9(4) COMP.
```

DCLGEN allows you to generate your table and view definitions providing the flexibility and benefits shown here.



To work correctly in an application program, some changes have to be made to a SELECT statement.

The first thing to consider is where the data that is retrieved by Db2 is going to be stored in the program. This data will be stored in a host variable.



Host variables are distinguished from columns by preceding them with a colon (:).

A host variable is not a column name but a working COBOL data item or a PL/I declared variable.

The easiest way to define host variables is to use a DCLGEN for the table since Db2 generates one variable for each column in the table. It never makes a mistake. The variable is always the correct data type, length, and so on.

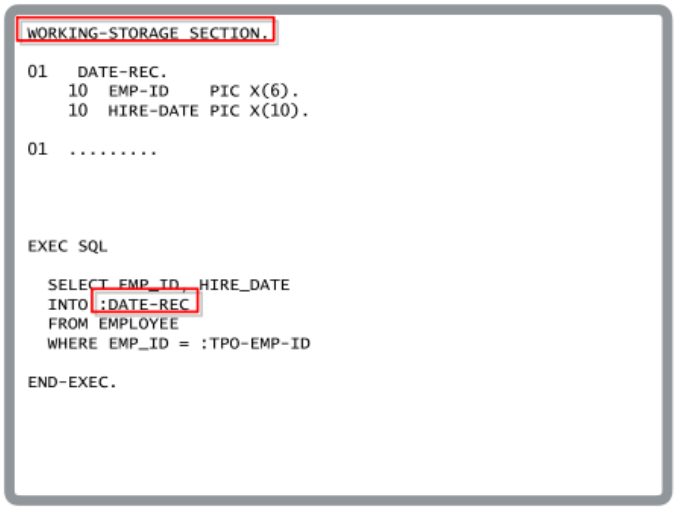
```
EXEC SQL
 SELECT EMP_ID,NAME
 INTO :EMP-ID, :NAME
 FROM EMPLOYEE
 WHERE EMP_ID = :TP0-EMP-ID
END-EXEC
```

This type of SELECT is sometimes called a SINGLETON SELECT. The syntax shown here is used for SELECT statements that return only one row.

If many rows are returned during execution, an SQLCODE-811 is passed back from Db2. This should be handled by the program with, for example, an abend.

You must ensure there is only one EMP_ID that has the value of TP0-EMP-ID on the table.

The SELECT statements that return multiple rows will be covered in a later module.



```
WORKING-STORAGE SECTION.

01   DATE-REC.
    10  EMP-ID    PIC X(6).
    10  HIRE-DATE PIC X(10).

01  .........



EXEC SQL

  SELECT EMP_ID, HIRE_DATE
  INTO :DATE-REC
  FROM EMPLOYEE
  WHERE EMP_ID = :TPO-EMP-ID

END-EXEC.
```

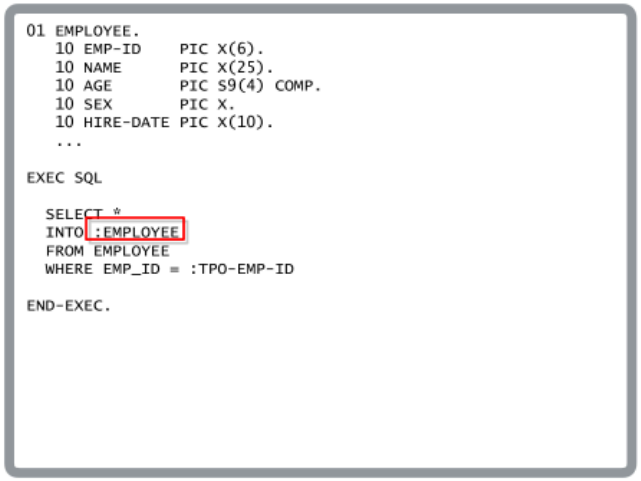
Values can be placed in a host structure, as shown in this COBOL example.

It is important to remember that the structure must be declared in the Working Storage Section.

```
01 EMPLOYEE.
   10 EMP-ID    PIC X(6).
   10 NAME      PIC X(10).
   10 AGE       PIC S9(4) COMP.
   10 SEX       PIC X.
   10 HIRE-DATE PIC X(10).
   ...

EXEC SQL

  SELECT EMP_ID, NAME
  INTO :EMPLOYEE.EMP-ID,
       :EMPLOYEE.NAME
  FROM EMPLOYEE
  WHERE EMP_ID = :TPO-EMP-ID

END-EXEC.
```

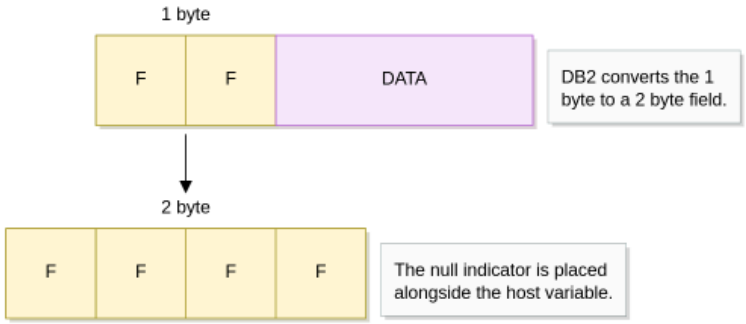
If the host variable you are using is in a structure, as shown here, and you want to fully qualify it, precede the variable name with the structure name and separate the two by a period (.).

Note: The syntax is the same for both PL/I and COBOL.



```
01 EMPLOYEE.
   10 EMP-ID    PIC X(6).
   10 NAME      PIC X(25).
   10 AGE       PIC S9(4) COMP.
   10 SEX       PIC X.
   10 HIRE-DATE PIC X(10).
   ...

EXEC SQL

  SELECT *
  INTO :EMPLOYEE
  FROM EMPLOYEE
  WHERE EMP_ID = :TPO-EMP-ID

END-EXEC.
```

The technique shown here is common when selecting all the columns from a table. The host structure must match the column definitions exactly, and be in the same order. This is not a good programming standard to use as errors can occur when table changes occur.



If the column selected can contain null values, being able to check whether a null has been passed back in the program would be convenient. This is done by using a null indicator.

A null is actually an extra byte tacked onto the column that has X'FF' in it, if the column is set to NULL.

X'FF' is a binary numeric value of -1; therefore, a null indicator is:

`PIC S9(4) COMP` in COBOL
`FIXED BIN(15)` in PL/I

```
01 NULL-IND                PIC S9(4) COMP.
        .  .
        .  .
 EXEC SQL

  SELECT EMP_ID, COMMISSION
  INTO  :EMPLOYEE.EMP-ID ,
        :EMPLOYEE.COMMISSION :NULL-IND
  FROM   EMPLOYEE
  WHERE  EMP_ID = :TPO-EMP-ID

 END-EXEC.

 IF NULL-IND = -1
  Then there was nothing in COMMISSION, so do something!
 END-IF.
```

Here is some code that checks for the NULL-IND=-1. It is possible to use any variable names you like, because it is not the name but the data type that is important.

Notice the syntax. There is no comma (,) between the variable and the null indicator. Spaces are ignored.

```
01  NULL-ARRAY.
   10 NULL-IND PIC S9(4) COMP OCCURS 2 TIMES.
*
01  DCLCUSTOMER.
   10 NAMEID        PIC X(8).
   10 SEARCH-NAME   PIC X(35).
          .
      .
EXEC SQL
   SELECT *
   INTO  :DCLCUSTOMER :NULL-IND
   FROM CUSTOMER
   WHERE NAME-ID = :TPO-NAME-ID
END-EXEC.
```

Take a look at this code. You can also use null indicators with host structures, but the technique is a little different.

It is important that each element of the array is examined for -1.

Output to the application		Input to DB2	
0 (zero), or positive value	Specified value is stored	0	Non-null value
-1	Null value	-1, -2, -3, -4 and -6	Null value
-2	A numeric conversion error has occurred	-5	If extended variables are enabled, then this specifies a DEFAULT value, otherwise it represents a null value.
-3	No value was returned	-7	If extended variables are enabled, then this specifies an UNASSIGNED value, otherwise it represents a null value.

As well as the -1 value, there are several other values that can be assigned to the indicator variable discussed on the previous pages. These can all be tested and appropriate action taken.

```
EXEC SQL
   DELETE FROM EMPLOYEE
   WHERE EMP_ID = 1000
END-EXEC.

EXEC SQL
   DELETE FROM EMPLOYEE
   WHERE EMP_ID = :EMP-ID
END-EXEC.
```

Db2 DELETEs or UPDATEs in an application program can be coded similarly to SINGLETON SELECTs.

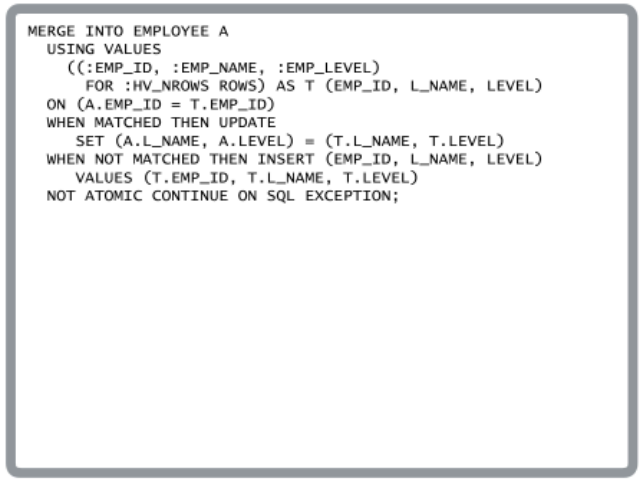
Remember SQL is a set language, so all the rows that match the WHERE clause will be updated or deleted.

Shown here are two DELETE statements; one uses a literal and the other a host variable to provide the predicate comparison value :EMP-ID.

```
EXEC SQL
   UPDATE EMPLOYEE
   SET NAME = "FIRED"
   WHERE EMP_ID = 1000
END-EXEC.

EXEC SQL
   UPDATE EMPLOYEE
   SET NAME = :NEW-NAME
   WHERE EMP_ID = :EMP-ID
END-EXEC.
```

An UPDATE statement is similar. Values can be set to values supplied as literals or in host variables.

The number of rows updated or deleted is placed in the field SQLERRD(3), in the SQLCA.



```
MERGE INTO EMPLOYEE A
  USING VALUES
    ((:EMP_ID, :EMP_NAME, :EMP_LEVEL)
      FOR :HV_NROWS ROWS) AS T (EMP_ID, L_NAME, LEVEL)
  ON (A.EMP_ID = T.EMP_ID)
  WHEN MATCHED THEN UPDATE
     SET (A.L_NAME, A.LEVEL) = (T.L_NAME, T.LEVEL)
  WHEN NOT MATCHED THEN INSERT (EMP_ID, L_NAME, LEVEL)
     VALUES (T.EMP_ID, T.L_NAME, T.LEVEL)
  NOT ATOMIC CONTINUE ON SQL EXCEPTION;
```

MERGE updates a target table with the specified input data if the row exists or will INSERT if the row does not exist. This removes the programming requirement to check for the existing of a row before determining what processing to use (INSERT or UPDATE).

In the example shown here, if EMP_ID in the source table matches the host variable EMP_ID, then that row's L_NAME and LEVEL values will be updated. If they don't match then a row will be inserted using the host variable values.

You can insert the number of rows that are specified in the host variable NUM-ROWS by using the following INSERT statement:

```
EXEC SQL
  INSERT INTO DSN8810.ACT
    (ACTNO, ACTKWD, ACTDESC)
    VALUES (:HVA1, :HVA2, :HVA3 :IVA3)
    FOR :NUM-ROWS ROWS
END-EXEC.
```

The following FETCH statement retrieves 20 rows into host variable arrays that are declared in your program:

```
EXEC SQL
  FETCH NEXT ROWSET FROM C1
  FOR 20 ROWS
  INTO :HVA-EMPNO, :HVA-LASTNAME, :HVA-SALARY :INDA-SALARY
END-EXEC.
```

When performing multi-row processing using FETCH, INSERT, or MERGE statements, host-variable-arrays can be specified. These host variable arrays allow multiple values to be captured into an array element.

Summary

Manipulating Data and Components of an Application Program

An application program using embedded SQL requires extra structures to be defined.

This module looked at these structures and examined how they are defined in a COBOL or PL/I program.

You should now be able to describe how:

- You Can Determine the Success of SQL Execution Using SQLCODE and SQLSTATE
- Host Variables Are Used to Pass Data Between Db2 and an Application Program