INTERNAL TRAINING

DataFlux Expression Engine Language

Course Notes

DataFlux Expression Engine Language Course Notes was developed by Jeffrey D. Bailey. Additional contributions were made by Allen Cunningham.

DataFlux Expression Engine Language Course Notes

Course code DF_EEL, prepared date 24Aug2010. DF_EEL_001

Table of Contents

Course Description

To learn more…

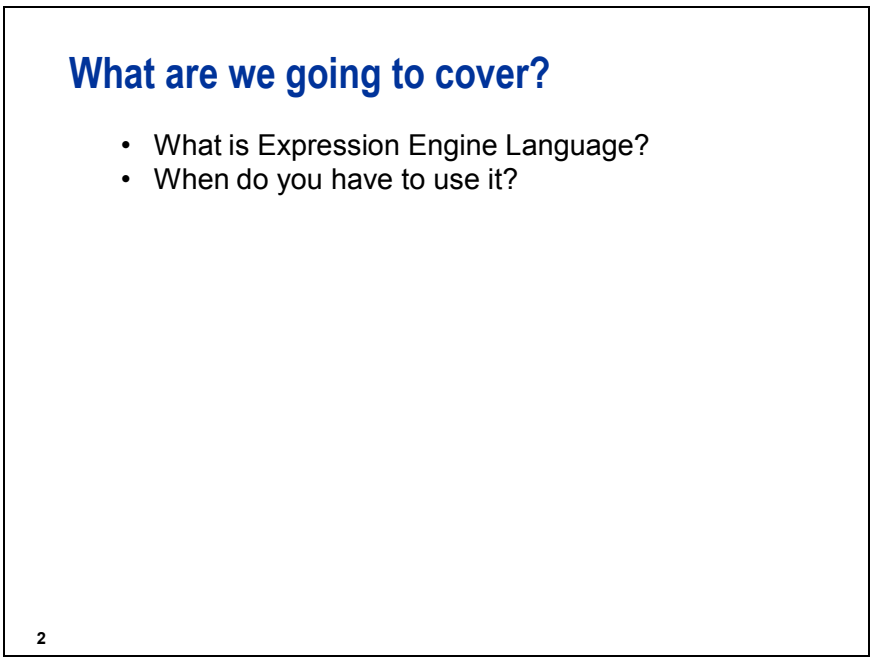
For information on other courses in the curriculum, contact the SAS Education Division at 1-800-333-7660, or send e-mail to training@sas.com. You can also find this information on the Web at support.sas.com/training/ as well as in the Training Course Catalog.

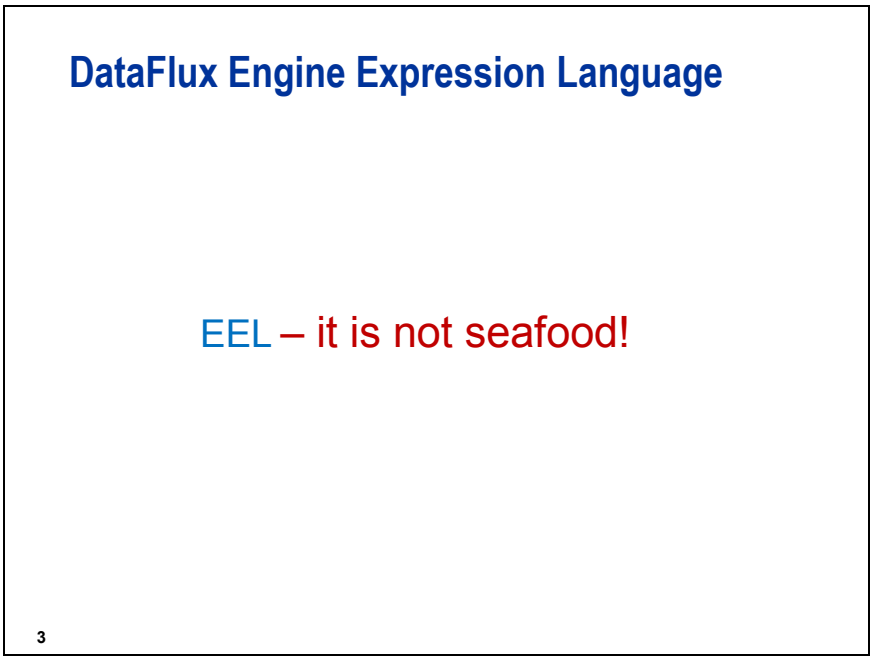
For a list of other SAS books that relate to the topics covered in this Course Notes, USA customers can contact our SAS Publishing Department at 1-800-727-3228 or send e-mail to sasbook@sas.com. Customers outside the USA, please contact your local SAS office.

Also, see the Publications Catalog on the Web at support.sas.com/pubs for a complete list of books and a convenient order form.

Chapter 1 Why Another Language?

1.1 Introduction





The DataFlux people don't like it when you call the language eel (think slimy, snake-like, animal that is quite good grilled). Call it Expression Language or E-E-ell.

DataFlux Engine Expression Language

EEL – it is a programming language!

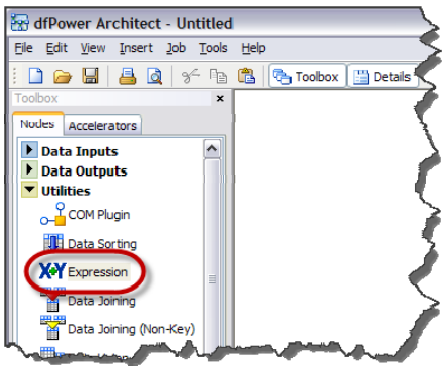
```
if isblank(`myName`) then
   empty_str_cnt = empty_str_cnt + 1

blank_str = isblank(`myName`)

if isnull(`myName`) then
   null_cnt = null_cnt + 1

null_str = isnull(`myName`)
```

4

Expression Language is a programming language. It is similar to Visual Basic. It is interpreted, not compiled. It is relatively easy to learn and simple to use.

DataFlux has a very rich feature set … So, why would you have to use a programming language?

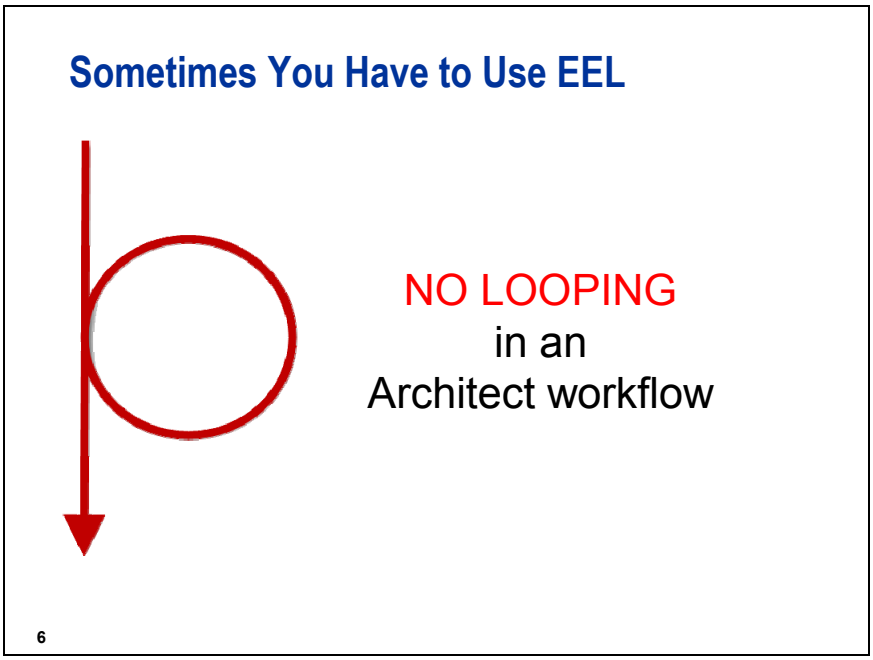
Expression Language is Very Flexible



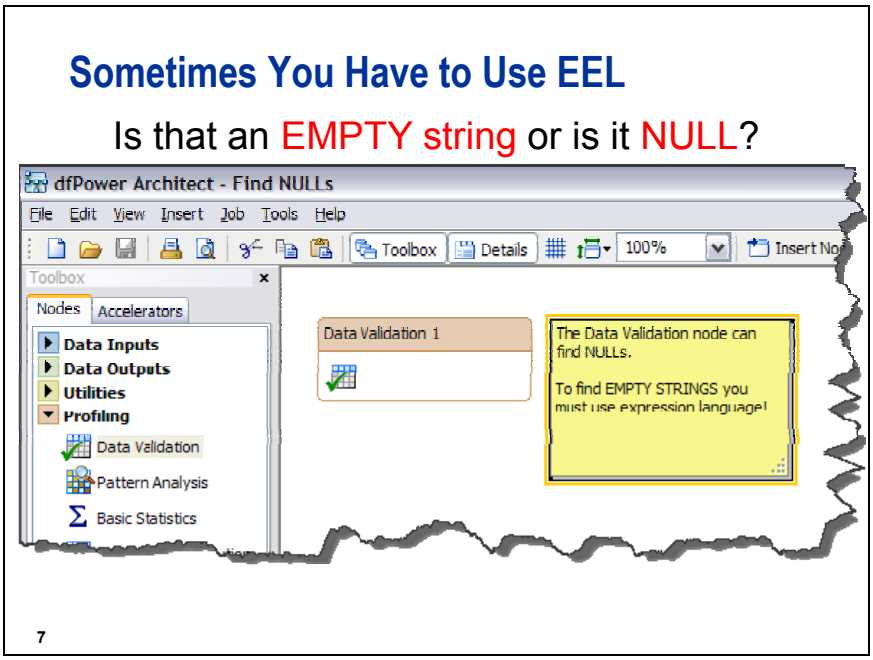
- You write the code
- Arrays
- Looping
- Precise NULL handling
- Set macro variables
- Custom metrics
- Objects

5

dfStudio Architect nodes serve a specific purpose. They are not designed for flexibility. That is not entirely true. There is one node that is designed for flexibility. That node is the Expression node.



There is no way to impliment looping in an Architect workflow. Fortunately, you can impliment looping – FOR and WHILE – using Expression Language.



You can filter NULLS using the Data Validation node. If you want to find the empty strings then you must use EEL.

Sometimes You Have to Use EEL

Complex String Manipulations
left()
right()
mid()
lower()
upper()
trim()
len()
replace()

8

Sometimes You Have to Use EEL

Create a new field!

```
STRING New_field
```

9

You can filter NULLS using the Data Validation node. If you want to find the empty strings then you must use EEL.

Sometimes You Have to Use EEL

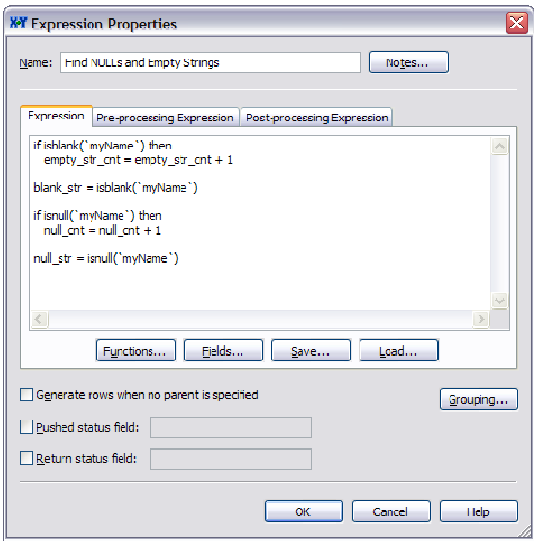
Manually assign a value to a field!

New_field = “Hello World”

10

You can filter NULLS using the Data Validation node. If you want to find the empty strings then you must use EEL.

Sometimes it is easier to code



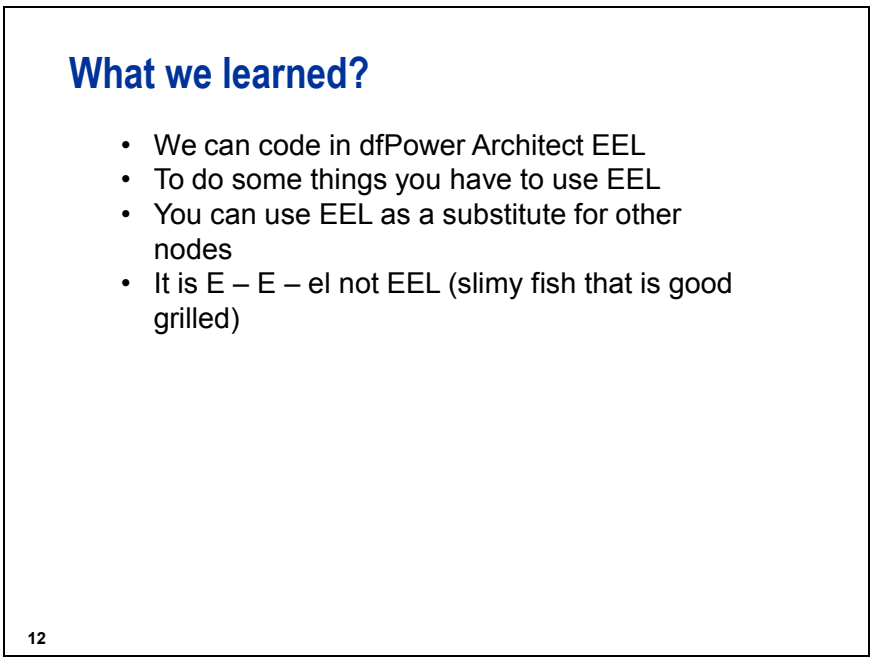
11

There are times when you can do something using the existing Architect nodes but you choose to code. There are many reasons for this: performance, ease, or coding simplifies the job.

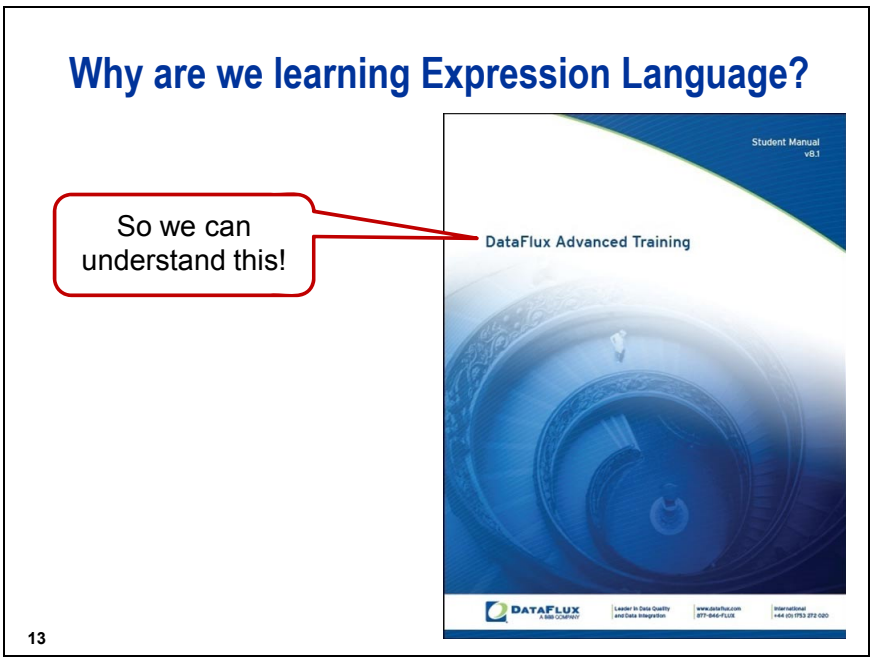
We will discuss this code in detail later. It is not complicated. There are a couple of IF statements using the isblank() and isnull() function to count the empty strings and NULLs found in the data. In addition to counting these occurences the code is setting boolean columns when it finds these things.

More about this later…

1.2 Chapter Summary



1.3 Final Thoughts



Last year, 2009, we traveled the world delivering a workshop similar to the one you are attending. The DataFlux Advanced Training course was included then but isn't being included now. The reason: our attendees had no clue what DataFlux Expression Language was. They had never heard of it.

The advanced training is all about EEL. Sadly, our students didn't learn much from that portion of the workshop. The content is great, but when you don't understand the underpinnings it is terribly difficult to learn. Here is what we want you to do.

1) Learn expression language. You are on the right path to do that today. This course isn't going to make you an expert but it will give you a head start in learning on your own. Pay special attention to Chapter 2. We have covered the Expression node completely. This is not documented very well and it takes time to grasp how it works, at least it took me a while to understand it. The purpose for this portion of the training is to give you a head start on steps 2 and 3.
2) Become familiar with the DataFlux Expression Language Reference Guide. This manual is installed as part of dfPower Studio. The language is covered in greater detailed than is possible during a half day workshop. This is a small language and the reference guide reflects that. At 116 pages it is quick to read. Make sure you try some of the code examples. Experimentation is a great way to learn this stuff.
3) Learn the techniques detailed in the DataFlux Advanced Training. The course notes, jobs and sample data are available on the DataFlux @ SAS Web site. Here is a link to the various training materials.

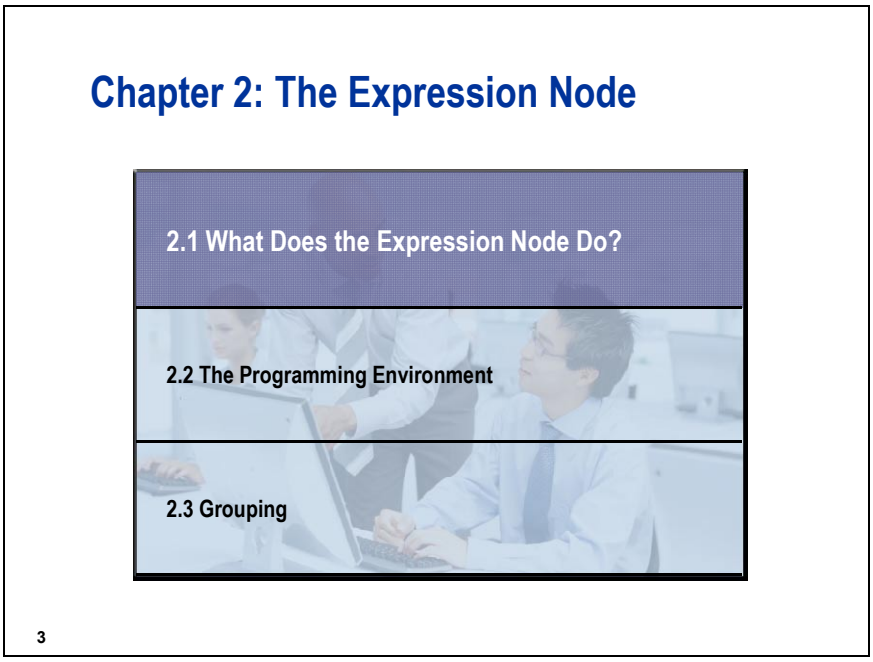
   http://sww.sas.com/dataflux/training/index.htm#materials_data_quality_platform

   You may need to scroll down a bit for the DataFlux Advanced Training materials.

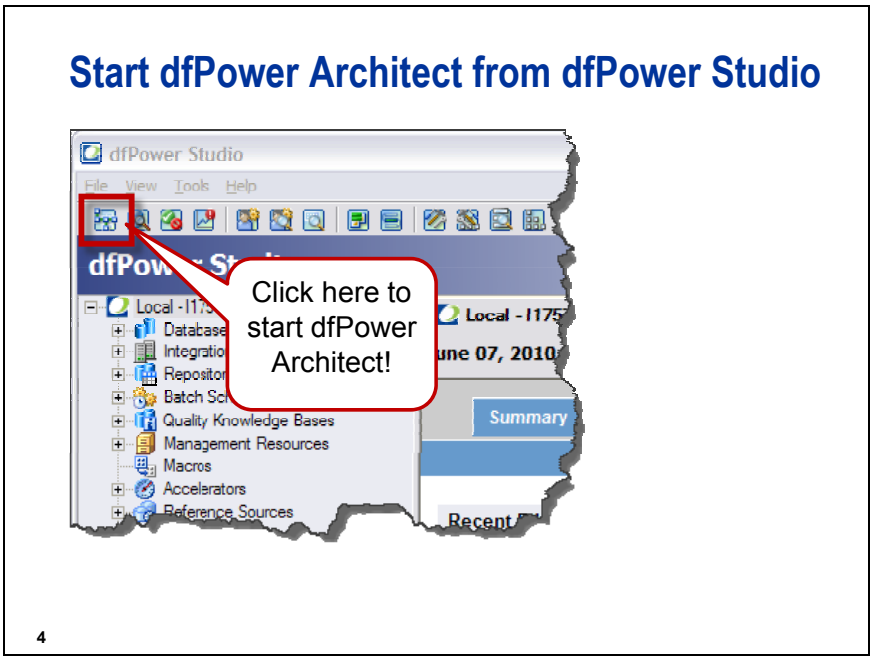
Once you understand EEL and the advanced topics from the advanced training you will be ready to handle the many data quality problems that can only be solved by writing custom code.

Chapter 2 The Expression Node

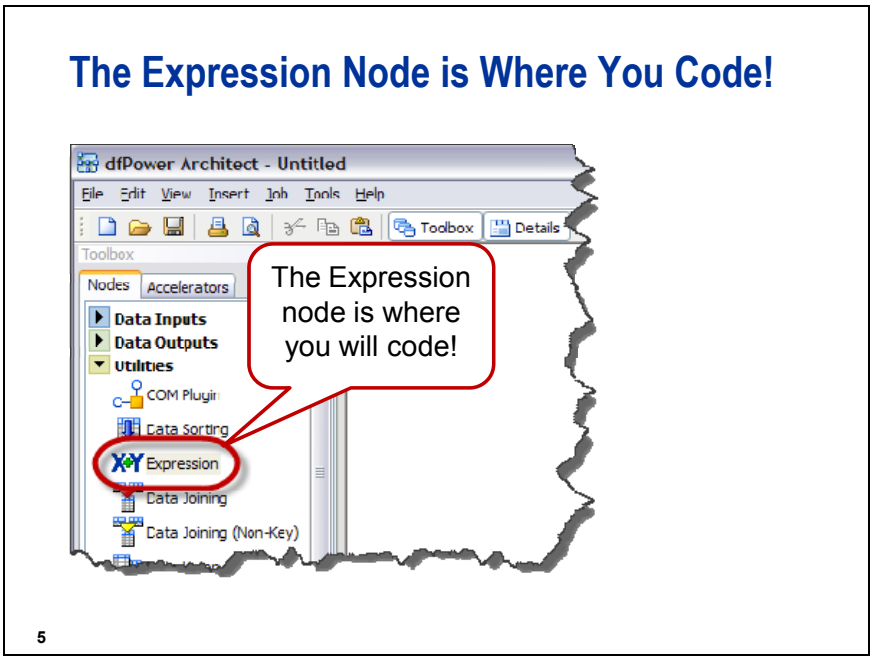
What Does the Expression Node Do?



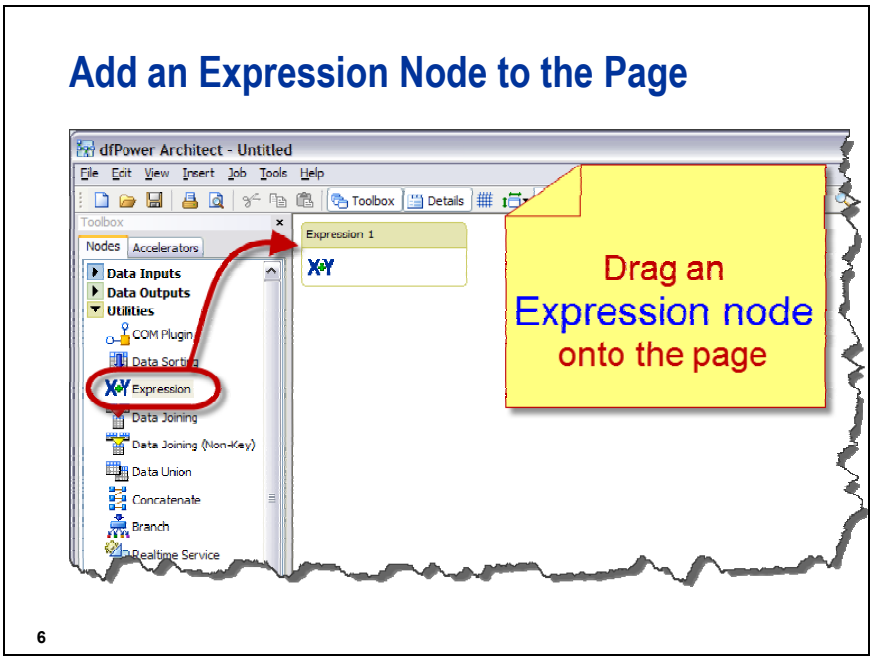
As we discussed in Chapter 1 there are times when you need to accomplish a task that doesn't have a dedicated node in dfPower Architect. Fortunately, the Expression node enables you to create your own custom nodes.



Let's open an Expression node and take a look at it. First, you must have dfPower Studio running. From there you will invoke dfPower Architect.

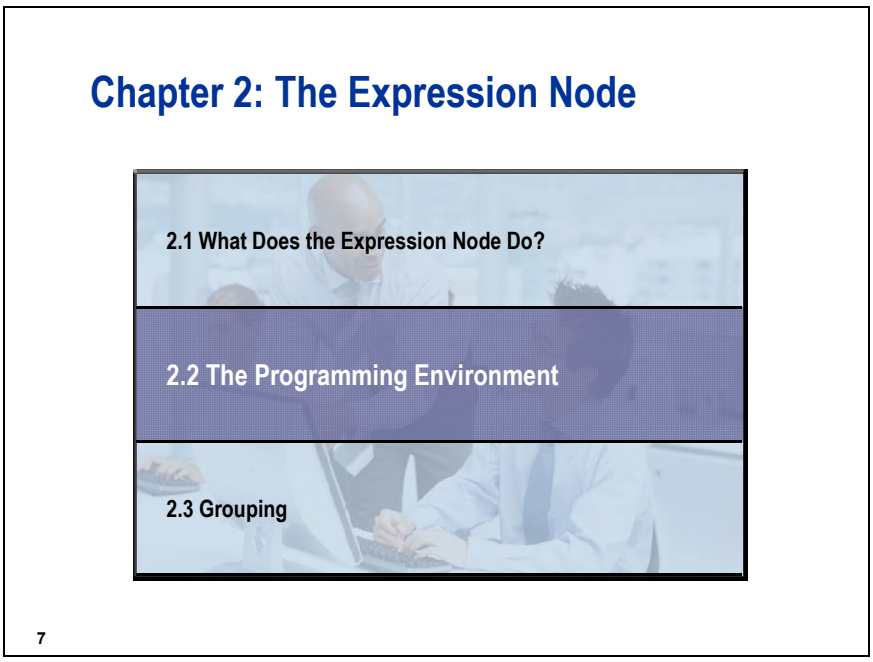


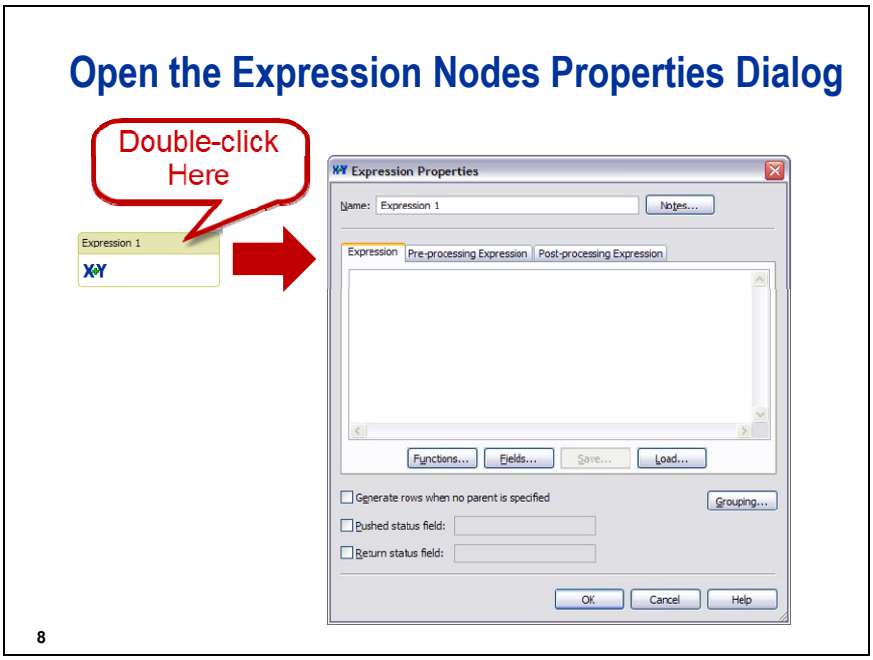
The available nodes are displayed on the left hand side of the dialog. The Expression node is utility node. Expand the Utilities category. You will find the Expression node there.



Add an Expression node to the page. You can drag it there.

2.1 The Programming Environment

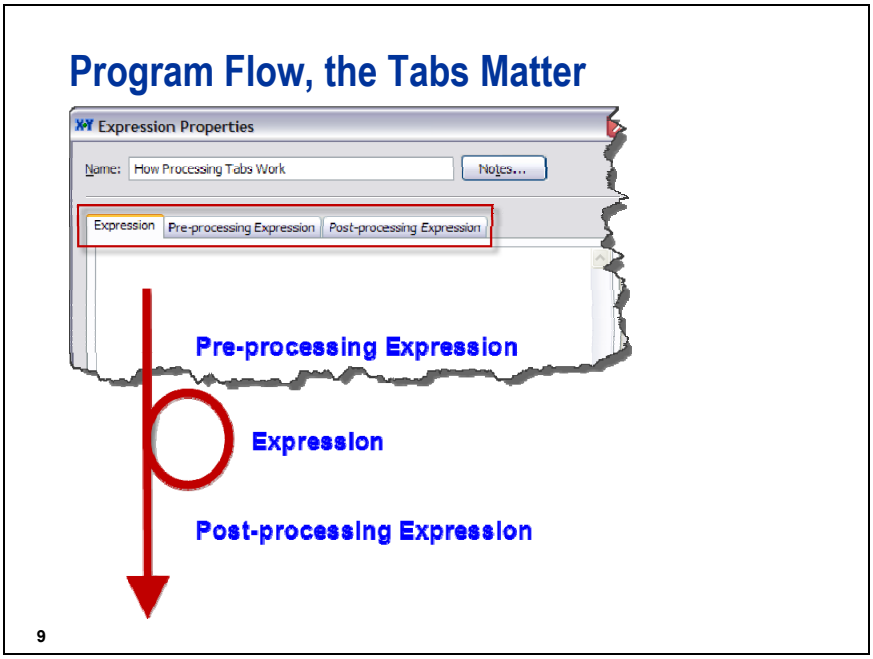




In order to see where you create your code, double-click on the Expression node.

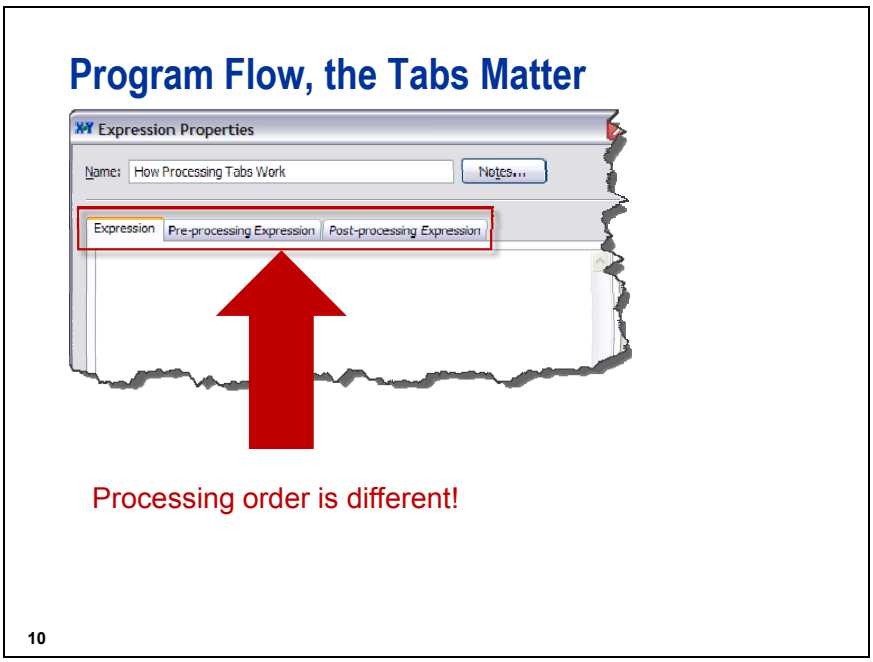
It is a good idea to give your node a descriptive name. This helps you in a couple of ways. First, it makes it easy to tell what the node is doing in the job flow. Second, it provides some insight into what the code actually does. Those reasons are very similar. The point is that you need to document what you are doing.

Once you are coding you will want to add comments to your code to assist in documenting exactly what it does. As you will see, it is easy to get lost in undocumented code. This is caused by the environment. You may have to enter code in many different places. Let's take a look.

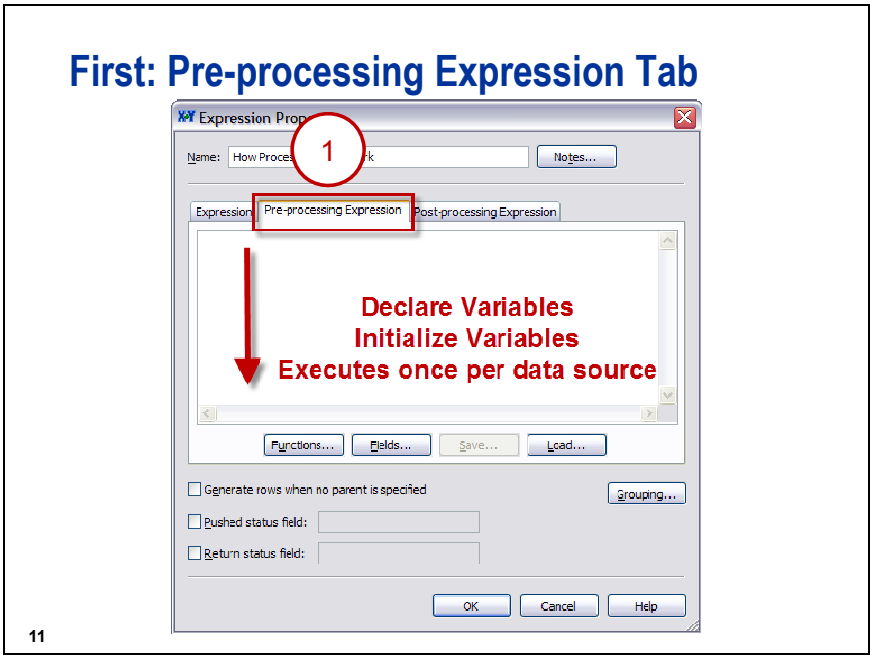


The code included on the Pre-processing Expression table executes first. Second, the Expression tab code is executed once for each input row. Third, the code from the Post-processing Expression is executed.

As you can see, where you put your code matters. It matters a great deal.

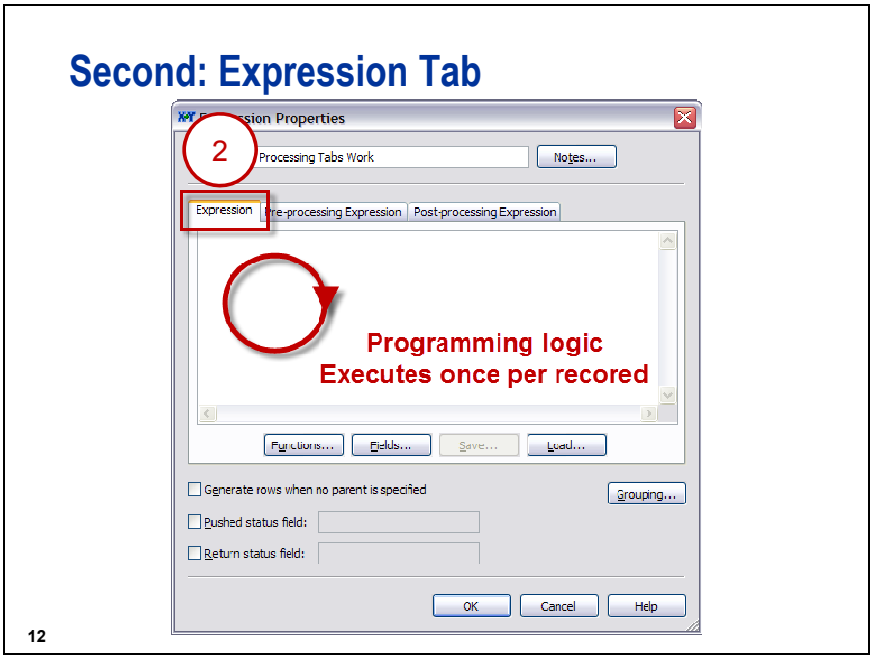


Unfortunately, the tabs are not listed in the order in which they process. You will likely enter code in the wrong tab. If your programs have “issues” then this should be one of the first places that you check.

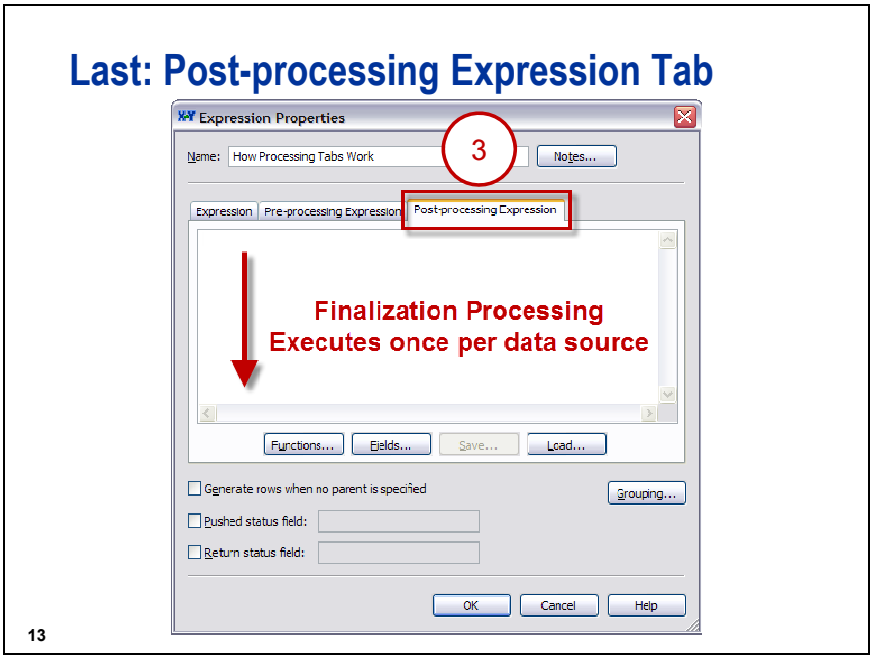


The code entered under the Pre-processing Expression is executed first. Pre-processing is commonly where you declare and initialize variables. If you are accessing database tables using Expression Language, then you may want to establish your connections here.

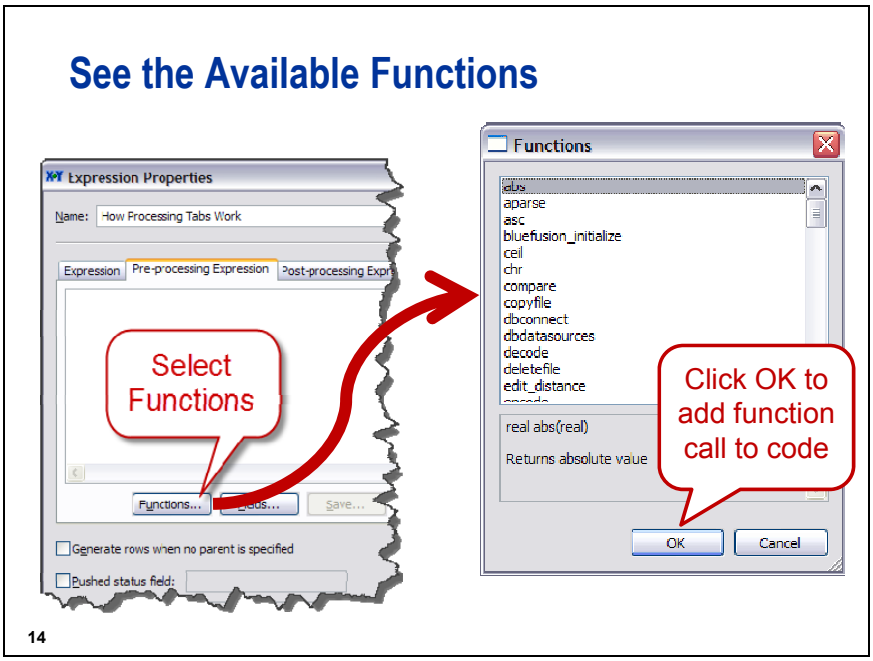
The code here executes once, at the beginning of the program. This happens before rows or records are read from the data source.



The Expression tab code is executed once per record read. This is where the real work usually takes place. Be careful here. If there are no records in your data source, or if the query from your SQL Node returns no rows, this code will not be executed. I don't know about you, but when I "learned" this it was surprising.

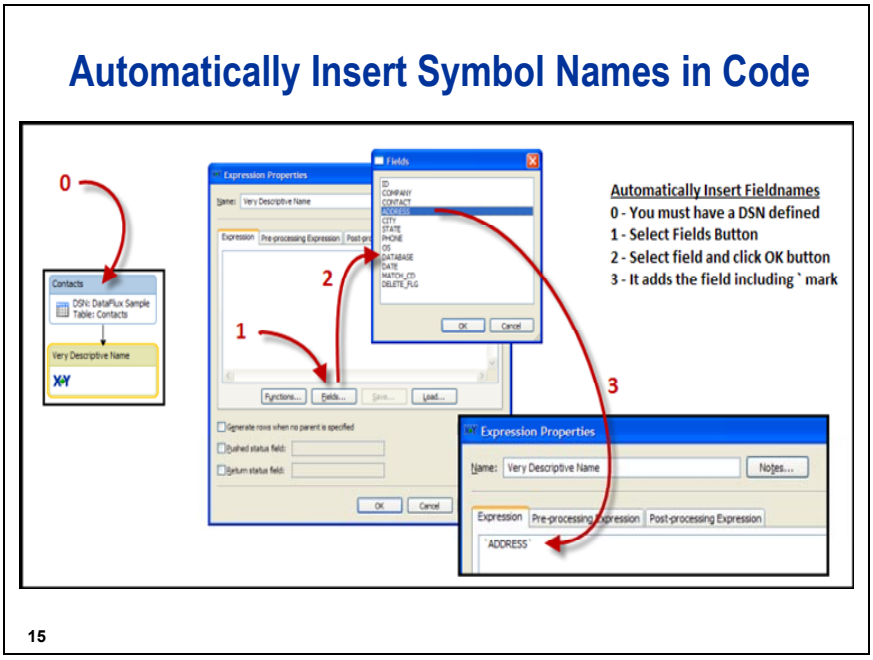


The Post-processing Expression is where you finalize your code. You can write statistics to the log, close your database connections, or do any other housekeeping that has to be done.



DataFlux's Expression Engine Language has a robust set of functions available for you to use in your code. Click on the Functions button and a list of available functions. You can high-light a function, in the

slide I have selected the absolute value function, and a description of how to call it and what the function does will be shown. When you select the OK button skeleton code will be inserted into your code.



Expression Language symbols are called variable in other languages. The Expression node helps you get your symbol names right.

In order to use the Fields button to insert a symbol you must have a data source name providing input to your Expression node. This does not mean that you must have a Data Source Node defined. It means that there must be a node that is a parent to the Expression Node which provides input data.

Here are the steps:

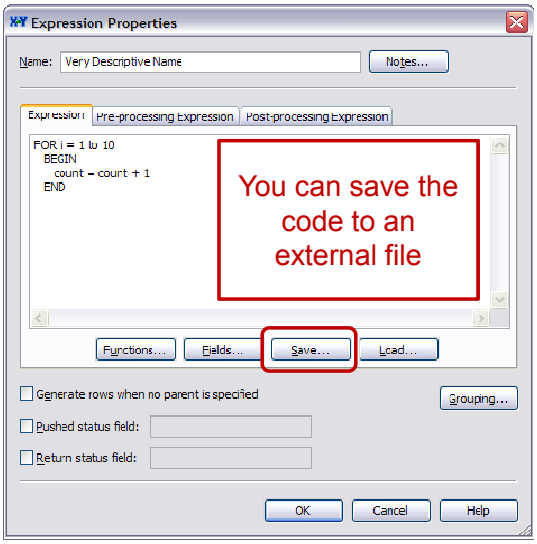
0 – have DSN information available to the Expression Node

1 – Select the fields button

2 – Select field and click the OK button

3 – The symbol will be added to the code

Save Code to a File



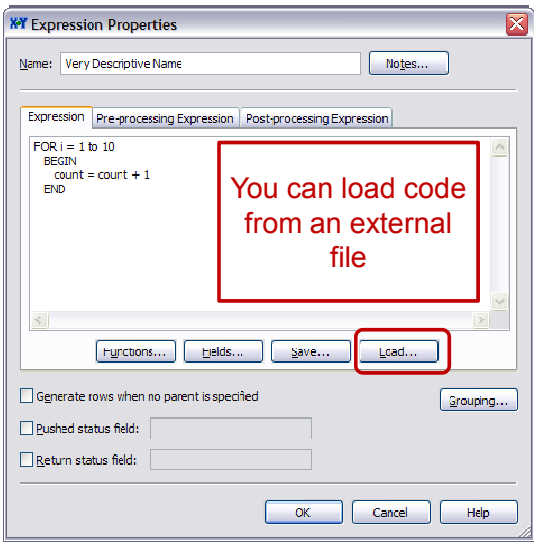
16

At some point you will write some code that you want to reuse or possibly back up. You can use the save button for this.

I have good news and bad news. First the good news. Saving code here will save the code in the Pre, Expression, and Post tabs. Now the bad news. It does not save the grouping code. You must do that as a separate step.

The dfPower Architect 8.1.1 the code is saved, by default, to C:\Program Files\DataFlux\dfPower Studio\8.1\Temp. The files will have an exp extension. From what I can tell, there is no way to successfully edit them. I tried using Ultra Edit and had problems. Your best bet is to only edit them in the Expression node.

Load Code from an External File

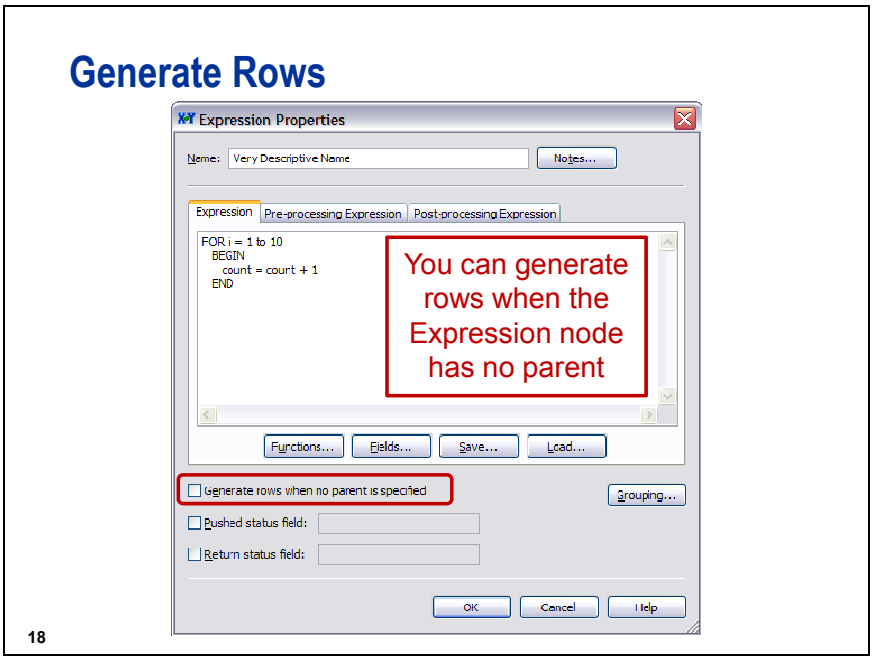


17

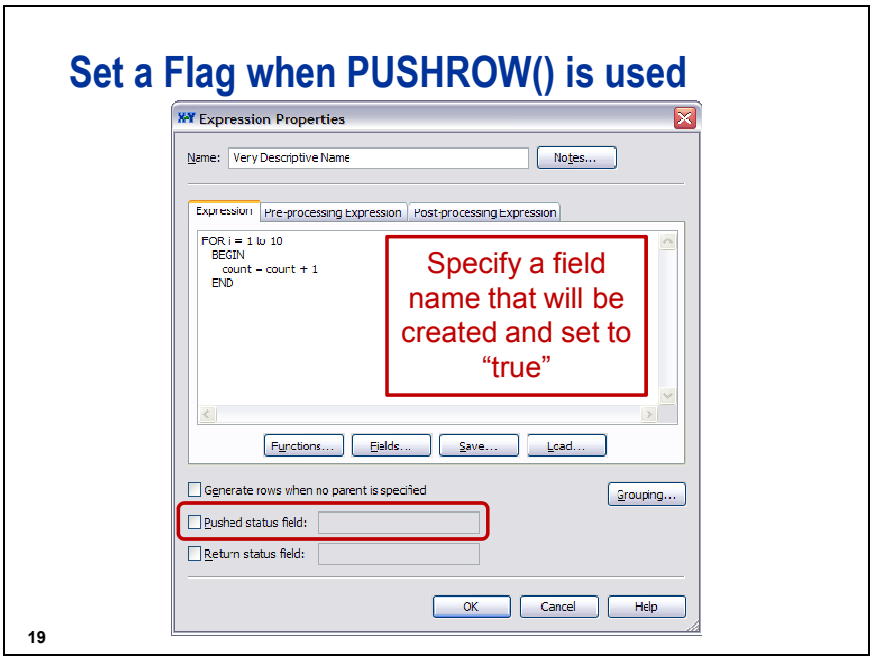
The Load button allows you to populate the Expression node with custom code. This comes in handy quite often.

Remember: by default the exported files (*.exp) are stored in C:\Program Files\DataFlux\dfPower Studio\8.1\Temp.

If you are using your SAS computer to create jobs for a consulting project you will want to ensure that your DataFlux files (Architect Jobs, expression code, etc.) are included in your backups. By default, these locations are not included. You will want to make sure that your customers are effectively backing up these files as well.

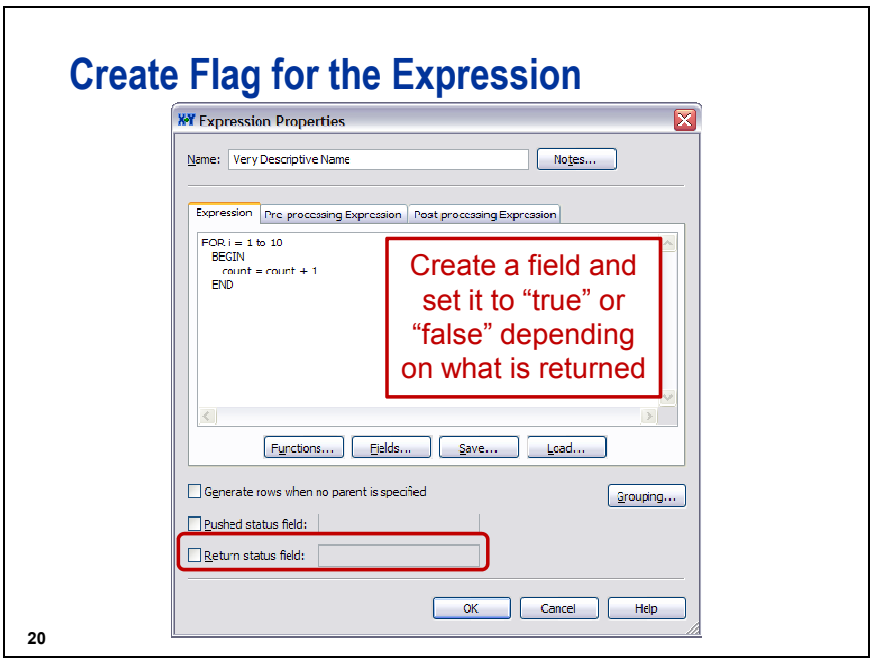


Selecting “Generate rows when no parent is specified.” allows you to create data rows when the Expression node has no parent in the Architect Job. This is quite useful for generating test data.



There is a function in Expression Language called pushrow(). The function pushes the current row onto a stack. The row will then be accessed by the next read operation. This in effect creates an extra row in the output.

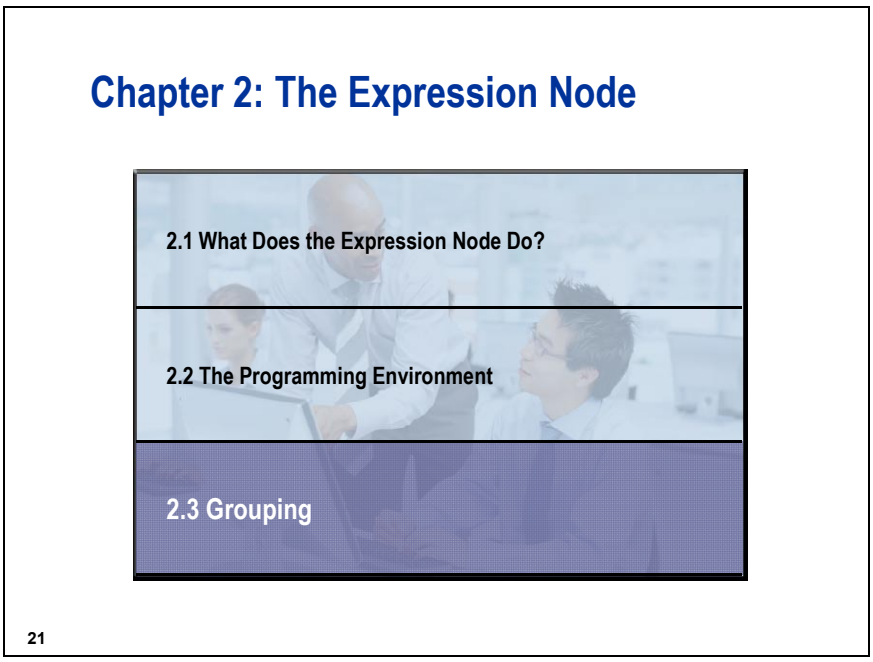
Selecting "Pushed status field" and entering a field name creates a new field in the output. A value of "true" means that the row was added to the output. A value of "false" means that the row was part of the input data. We will discuss this in detail later.

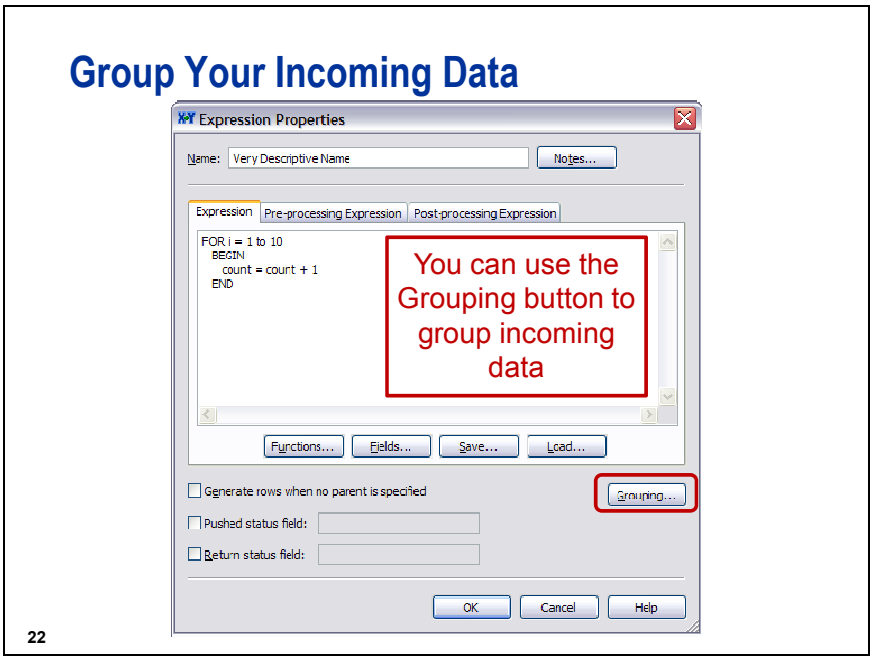


When an expression returns a value of “false” it isn’t included in the output. This behavior is very useful when you create complex filtering logic. There are sometimes that you want to see the rows that failed the test. That is where the return status field comes in.

Selecting “Return status field” and providing a field name includes the rows that failed the test in the output. You get to see these failed rows and do further processing on them.

2.2 Grouping

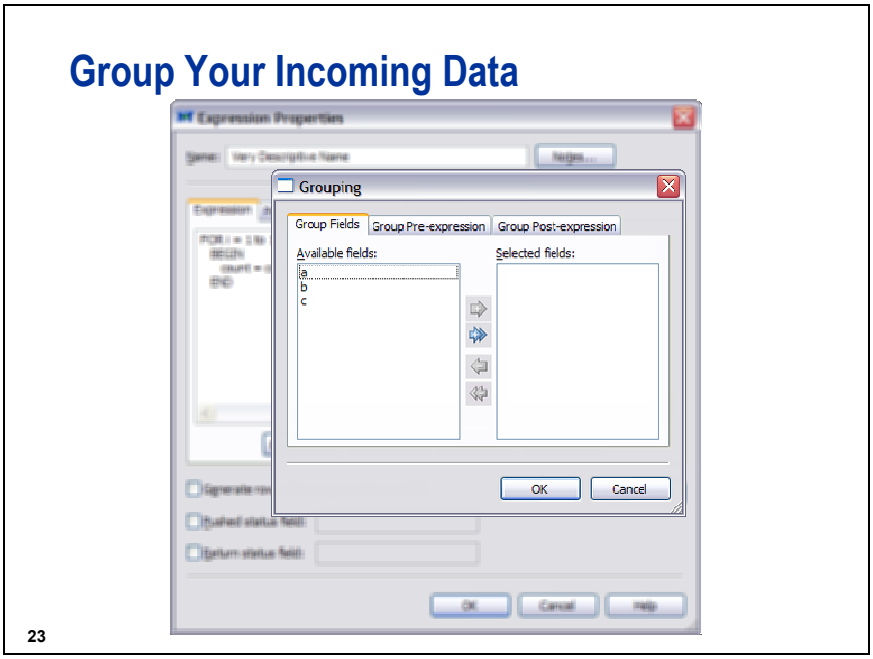




The Expression node is very powerful. You can do things with it that are impossible using any other node. As we discussed earlier, you use the Expression node to create one-of-a-kind, special, nodes. You can even do GROUP BY processing. Just like in SQL.

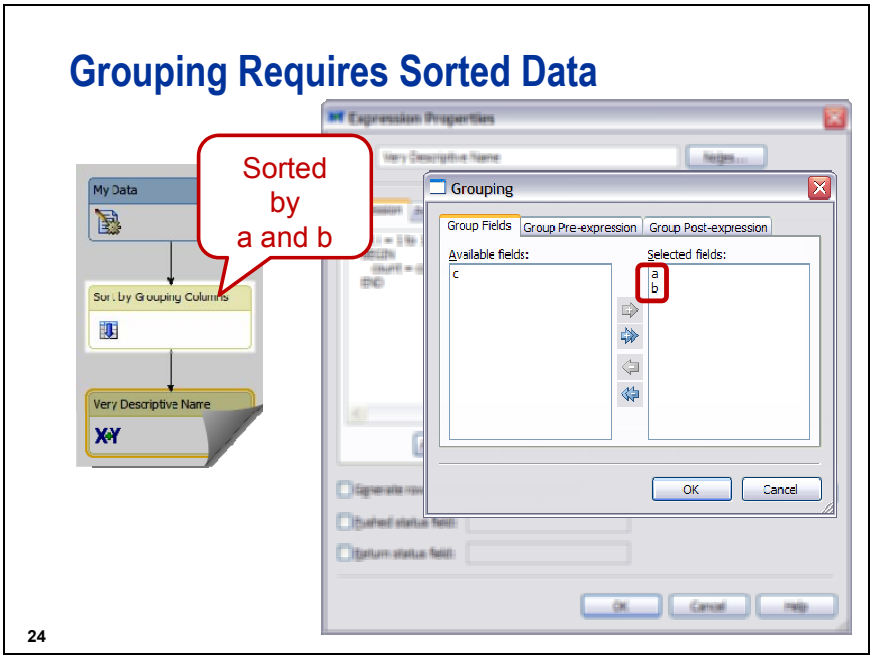
I find using the Grouping functionality is somewhat confusing.

Fortunately, I know Jim Hart. Jim has provided me with a great example of grouping in the Expressions node. He has written PROC SUMMAY in an Expression node. We will see that soon. Right now, let's just talk about the interface…



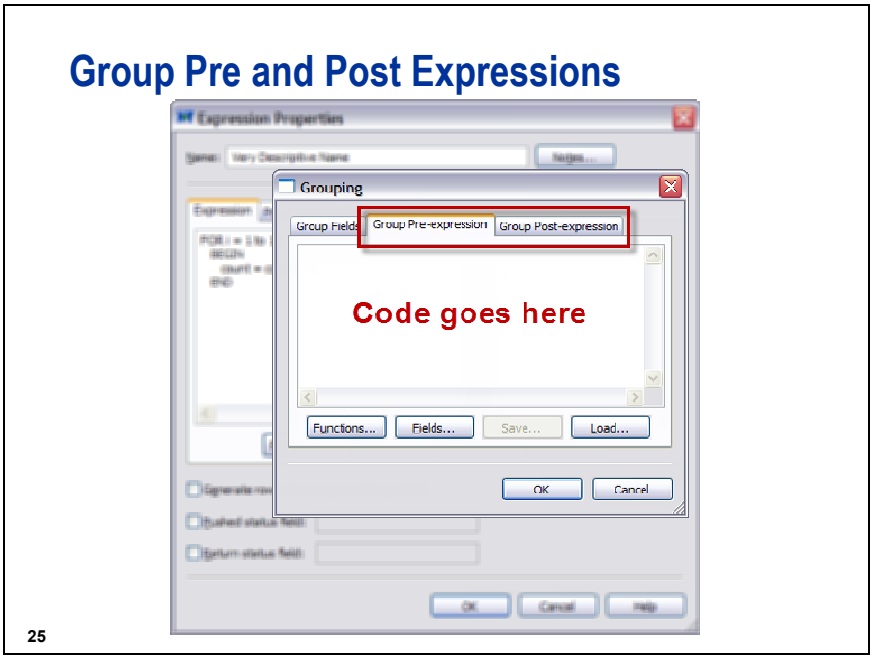
Clicking on the Grouping button brings up the Grouping dialog. The first thing that you will do is choose the fields that you want to group on. You highlight them and then use the arrows to move them into the Selected fields area on the right.

It isn't quite this simple. There is one thing that we need to do before we get started.



We must sort the data by our grouping fields. In this case we want to group by columns `a` and `b`. That means that we need to drag and drop a Data Sorting node onto the page. We need to configure that node to sort by `a` and `b`.

Notice that the those fields have been moved to the Selected fields. They will be used for group processing.

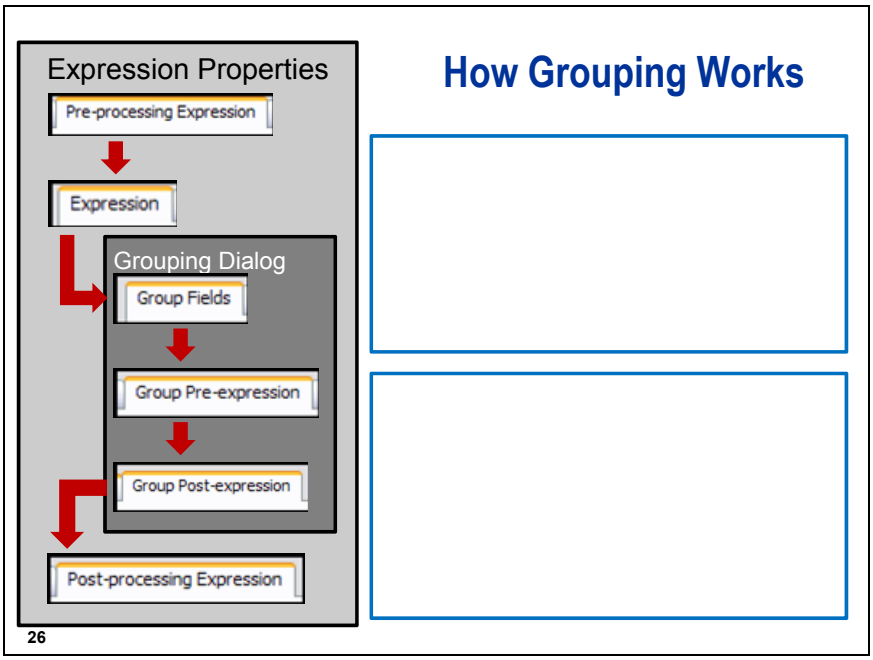


Let's take a look at the Group Pre-expression tab. The Group Post-expression is the exact same.

The Group pre-expression tab is where the code that runs at the beginning of each group. You will do things like initialize counters and sums to zero. You want those to restart at the beginning of group processing.

The Group post-expression tab is where you calculate statistics for the group. For example, let's say you are grouping by sales regions and you want to calculate average sales per region. That calculation would take place here. Once the calculations are complete, you will likely use the PUSHROW() function to add the row to the output.

The Functions, Fields, Save, and Load buttons are the same as we saw in the Expression tab. Well, there is one difference. The Save and Load buttons. If you are in the Group Pre-expression tab and save your code the post-expression code will not be saved to an external file. It is the same with load. You must save and load in each specific tab.



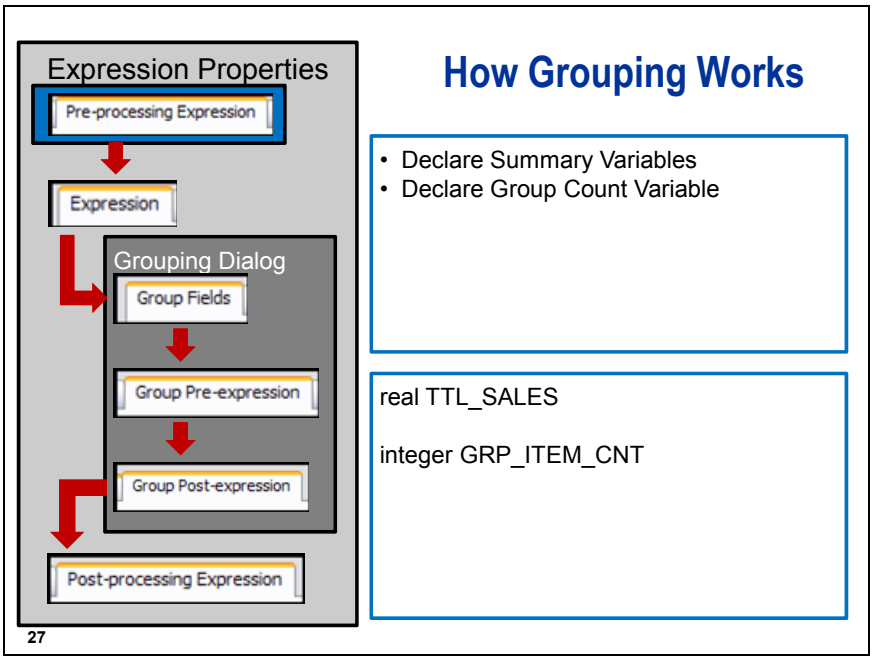
Let's take a look at how the Expression node will execute the code.

To the left we have the Expression Properties dialog and the Grouping dialog. We will step through these dialogs and tabs in the order that they execute.

In the upper right-hand box we will describe the type of processing that will take place. In the lower box we will show example code. We will display descriptions and code that would be entered into the highlighted dialog tab.

The example that we will look at was proved by Jim Hart, who works for DataFlux. It reads sales information and calculates sales by company and product number. In order for this to work we must sort the data prior to its arrival to the Expression node.

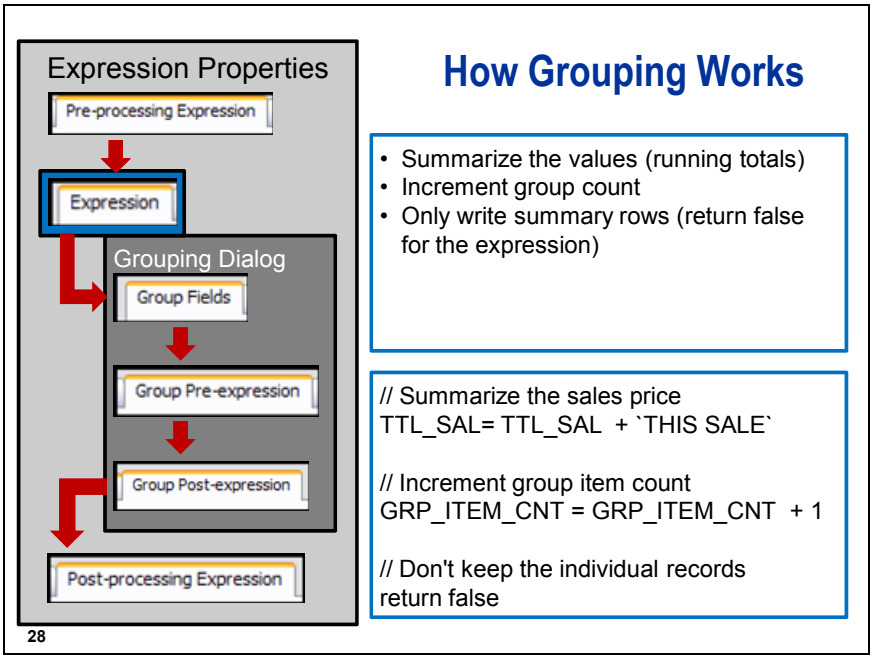
We will start with the Pre-processing Expression.



The Pre-processing Expression tab is a great place to define symbols (variables) and give them values.

Our job computes average sales by company and product. The TTL_SALES symbol will be used hold the sales for the group. In order to calculate the average we need to know how many members there are in the group. That is where the GRP_ITEM_CNT comes in.

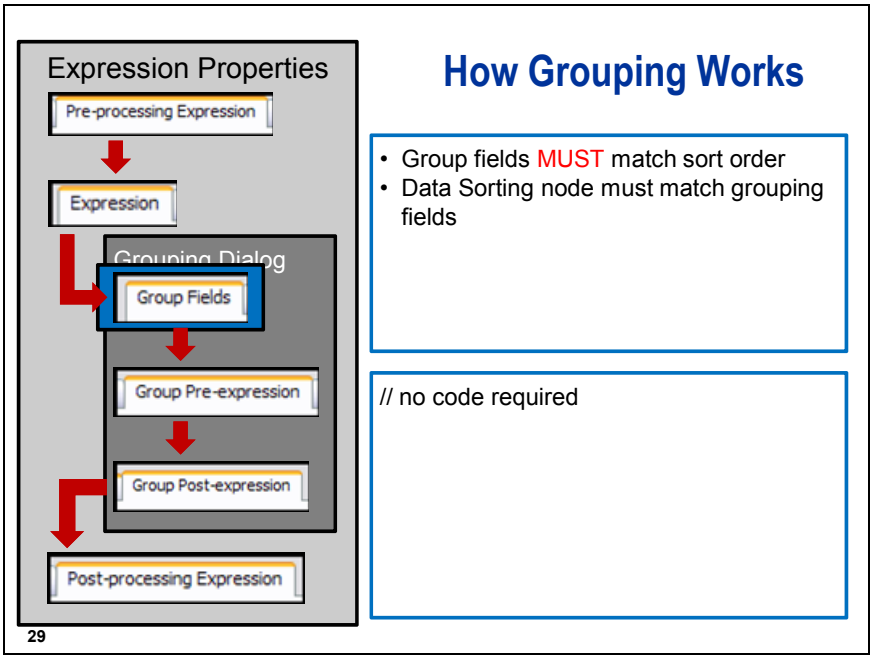
Notice that these symbols are not initialized here. We will see that shortly.



The code entered in the Expression tab is where the node loops through the input records. It is at this point that summary data and group item counts are tabulated.

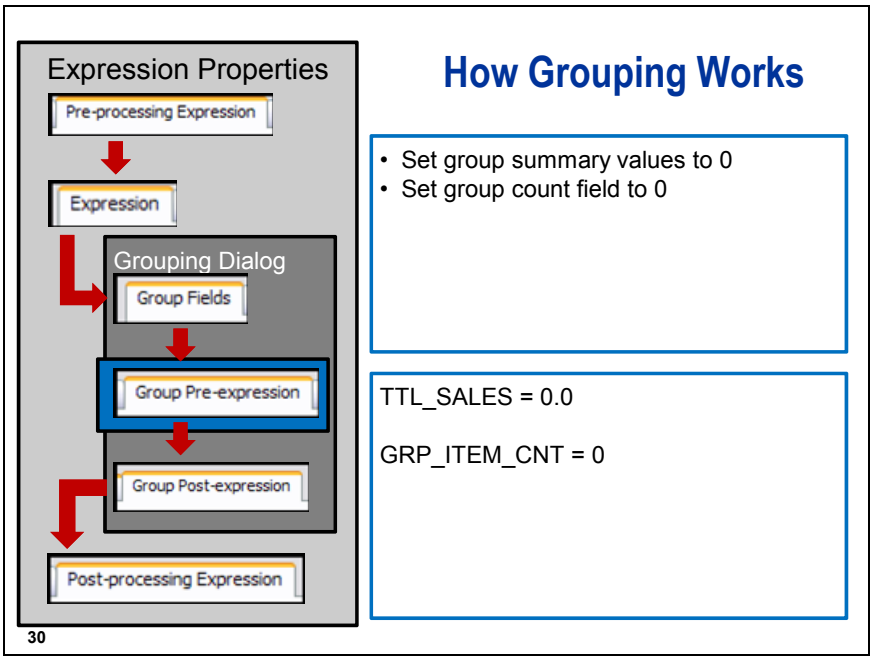
This is important. When an expression returns true (the default) then the record is written out. When creating summary data we don't want that to happen. We will set the return value to false. When we do this we must manually tell our expression to output the records. If we fail to do that then our output stream will be empty.

We will output the rows in just a moment.

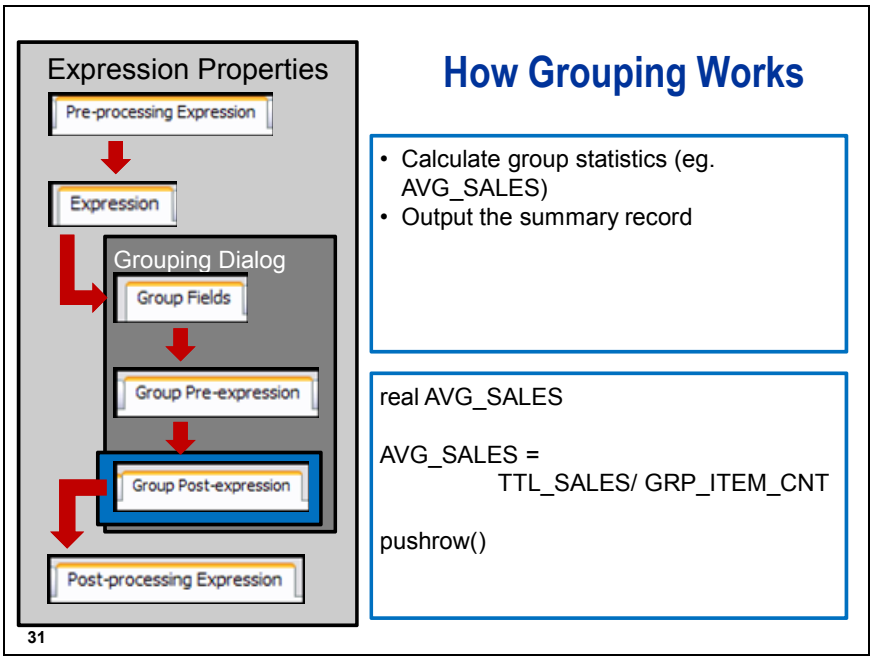


This is very important. The data must be sorted by the group by fields prior to being read by your Expression node. This is accomplished using a Data Sorting node. In the Group Fields tab, make certain that your fields are listed and in the proper sort order.

You don't need to code anything here. In fact, you cannot write code here.



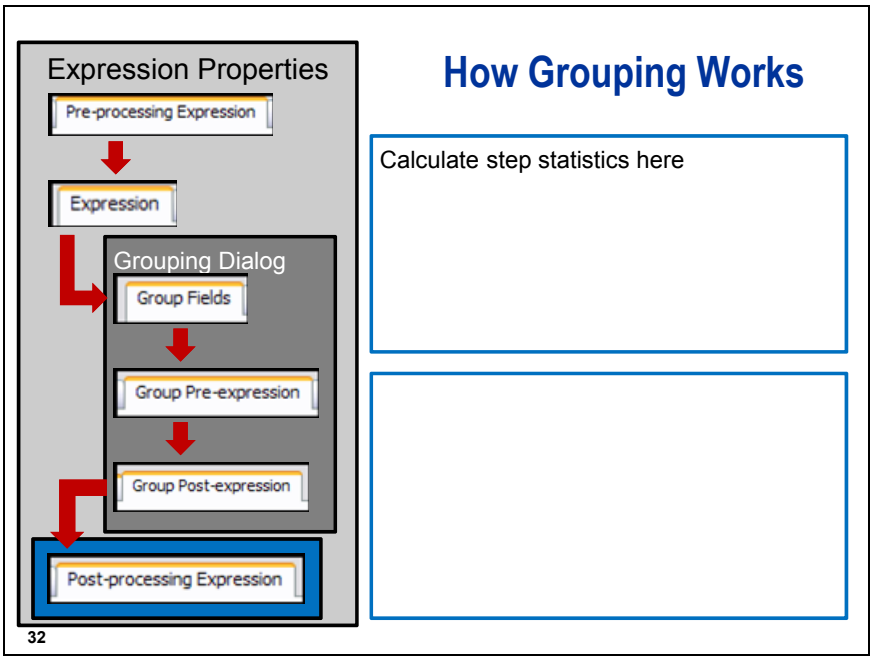
The Group Pre-expression Tab is used for initializing your summary and count fields. Remember, this will need to happen every time there is a change in the group by fields.



The Group Post-Expression is executed at the end of group by processing. This is where you will calculate the group by statistics and write the record to the output stream.

In this example we take the total sales for the company and product combination and divide it by the number of items (records) in the group.

Once the calculations are complete the pushrow() function is called to write the record.



If you need to calculate statistics for the entire expression you can do them here. You may want to count the number of records read by the node. Or something similar.

2.3 Exercises

Open the "EEL Chapter 2 Sample 01" Architect job. Perform the specified tasks and answer the included questions.

1. Where are the symbols initialized?

2. Preview the Expression node. Without looking at the help, see if you can figure-out what the print functions does.

3. On the Expression Properties dialog, select the "Pushed status field" and enter "Push_Status" for the field and then rerun the job. What happens?

Open the named "EEL Chapter 2 Sample 02" Architect job. Perform the specified tasks and answer the included questions.

4. There is a problem with this job. It should produce data what you right click the Expression node and select Preview but it doesn't. What can you do to fix the problem? When you have the preview working, run the job.

5. The seteof() (on the Expression tab) function is sort of lonely down there. What do you think it does? Hint: Don't remove it and then run the job. It will cause a lot of trouble.

Open the named "EEL Chapter 2 Sample 03" Architect job. Perform the specified tasks and answer the included questions.

6. Explore the nodes in this job. Pay special attention to the Expression node and the Grouping dialog. Do you understand what the job is doing and why it works. Hint: don't just look at it and believe that you understand it. Run it, change it, rerun it, etc.

2.4 Solutions

Solutions to Exercises

1. The symbols (fields) are initialized in the Pre-processing Expression tab.

2. The print function writes to the Log when you preview a job.

3. A field named Push_Status is added to the output data source. The value is set to false since the row was not created via the pushrow() function. If a row is written as a result of the pushrow() function then the value will be true.

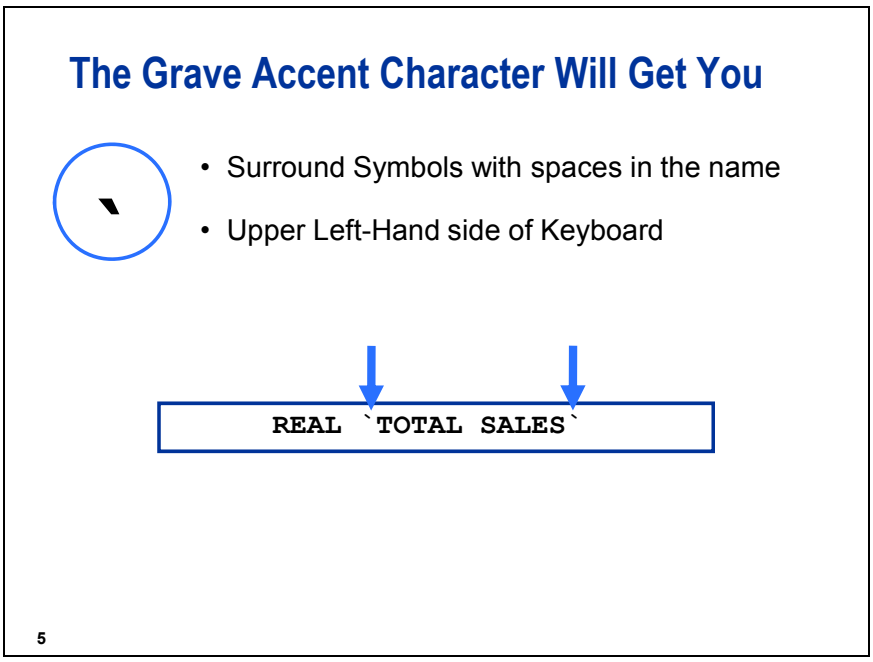
4. The problem with this job is that there is no data being read by the Expression node. To have the preview work you must select "Generate rows when no parent is specified." Once you select this the job will work.

5. The seteof() function tells the expression to stop processing. If you remove this function call then preview will work because it reads a set number of rows and then stops. You can change this number in dfPower Architect by going to Tools→Options and changing the value specified by "Number of rows in preview pane." When you run this job without executing seteof() the node will keep generating empty rows (infinite loop) that will eventually fill up a disk.

6. There is no real solution to this one. Please explore the code and ensure that you understand it. It is a fact that the best way to learn Expression Language is to experiment. This program provides a great playground on which to do this. Take advantage of the opportunity.

Chapter 3 The Language

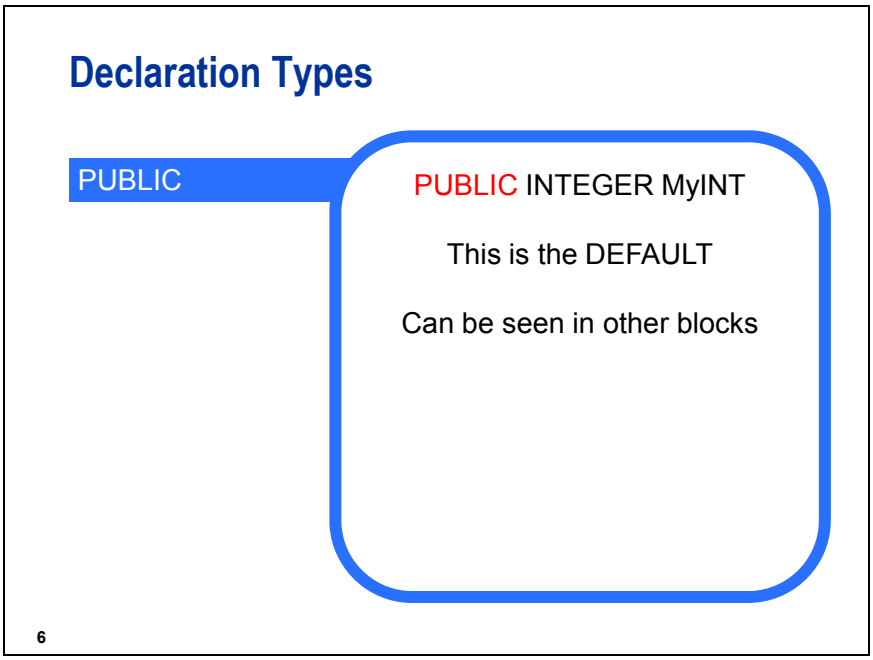
3.1 Declarations



This little character will cause you so much grief. I get so frustrated with it that I have created a rule for myself.

Never include a space in a symbol name.

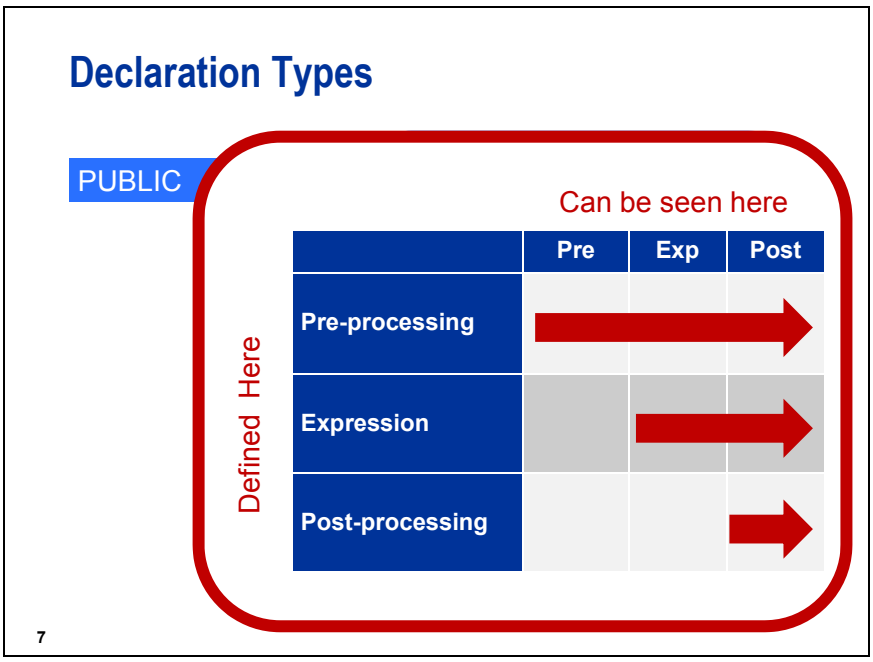
This saves a lot of aggravation. You may want to adopt this rule as your own.



If you don't specify the scope the default will be PUBLIC. That means if I don't include the keyword PUBLIC then the declaration is the same.

PUBLIC INTEGER MyINT = INTEGER MyINT

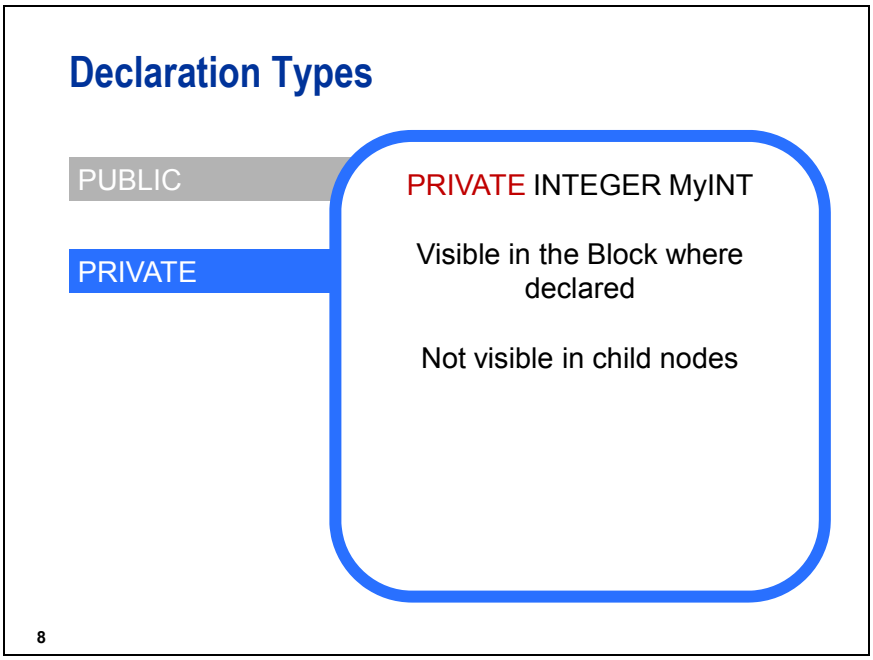
Public means that the variable can be seen in other expression blocks. Let's take a look at that.



Public symbols declared in the Pre-processing tab are available to any block.

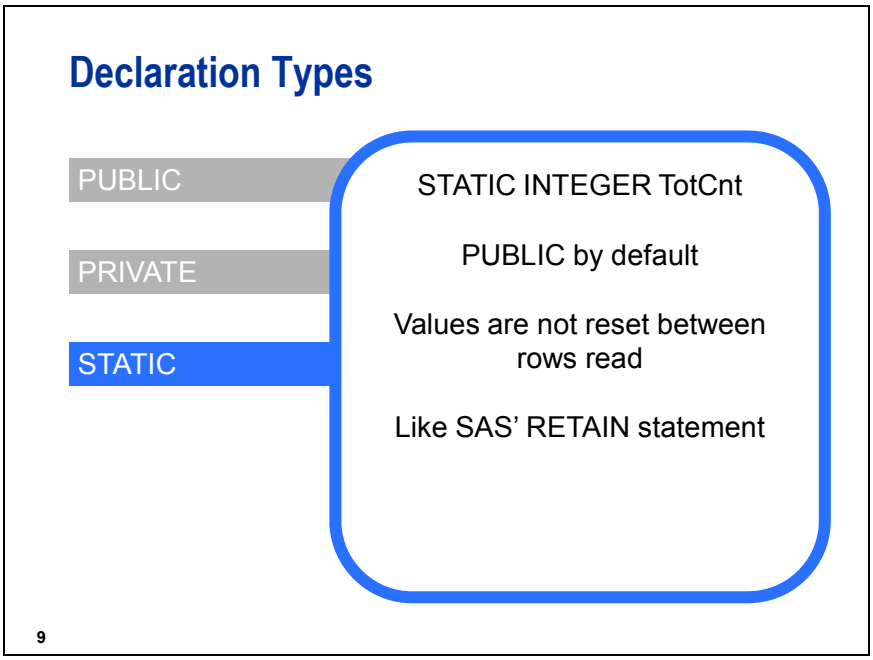
Public symbols declared in the Expression tab are available to the Expression and Post-processing blocks.

Public symbols declared in the Post-processing block are only available to the Post-processing block.



PRIVATE symbols are only visible in the block where they are declared. This allows you to use a symbol name in more than one block. Be careful with that, it can be very confusing.

Symbols defined as PRIVATE are not available to child nodes in the Architect Job.

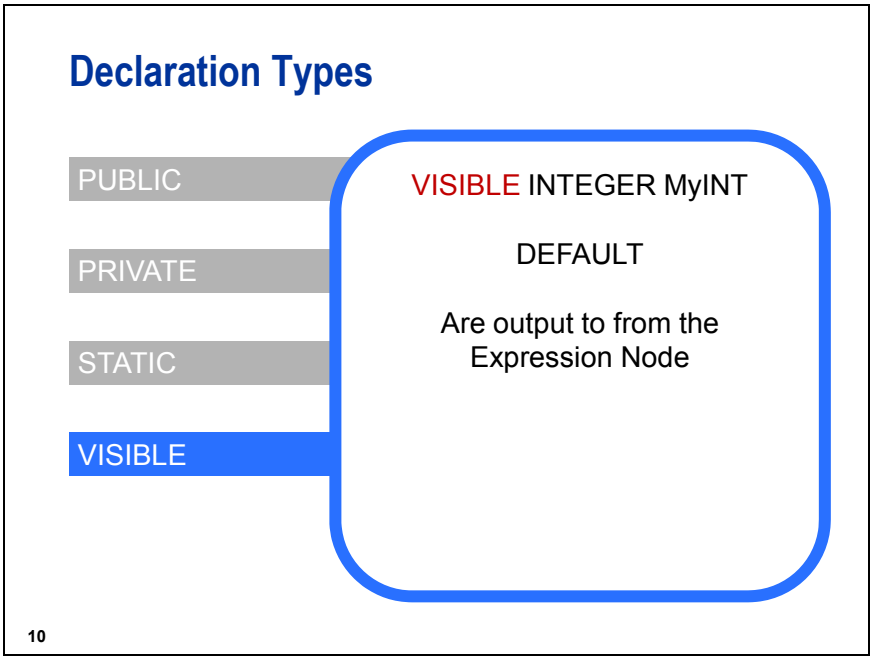


Static symbols are public by default. They can also be declared as private.

PRIVATE STATIC INTEGER TotCnt

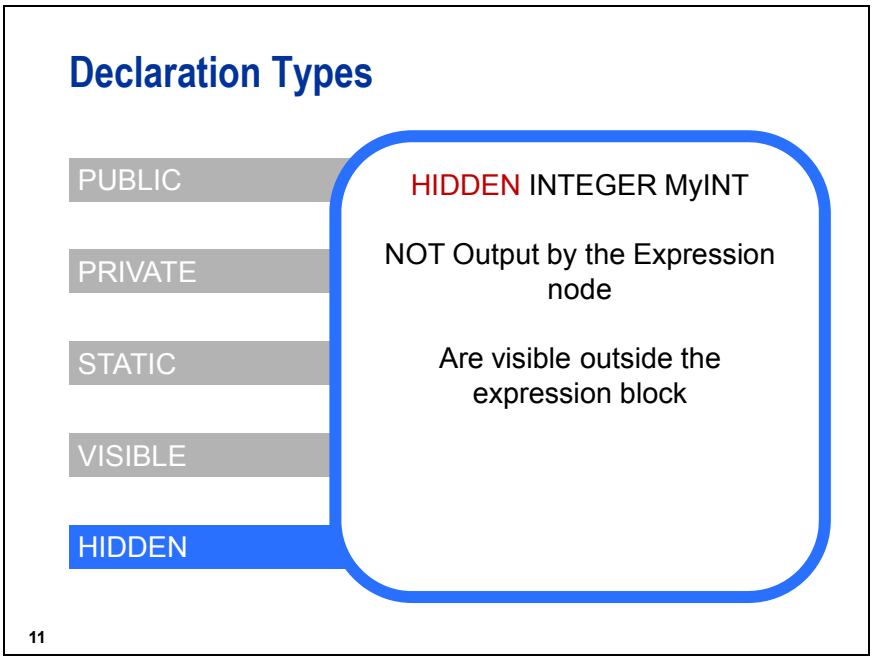
The symbol's values are not reset between rows. For the non-STATIC variable the values are reset to NULL when a new record is read.

This is very similar to the SAS RETAIN statement.



Declaring a symbol as VISIBLE ensures that it can be seen in child nodes. That means that the symbol in output from the Expression node.

VISIBLE is the default.



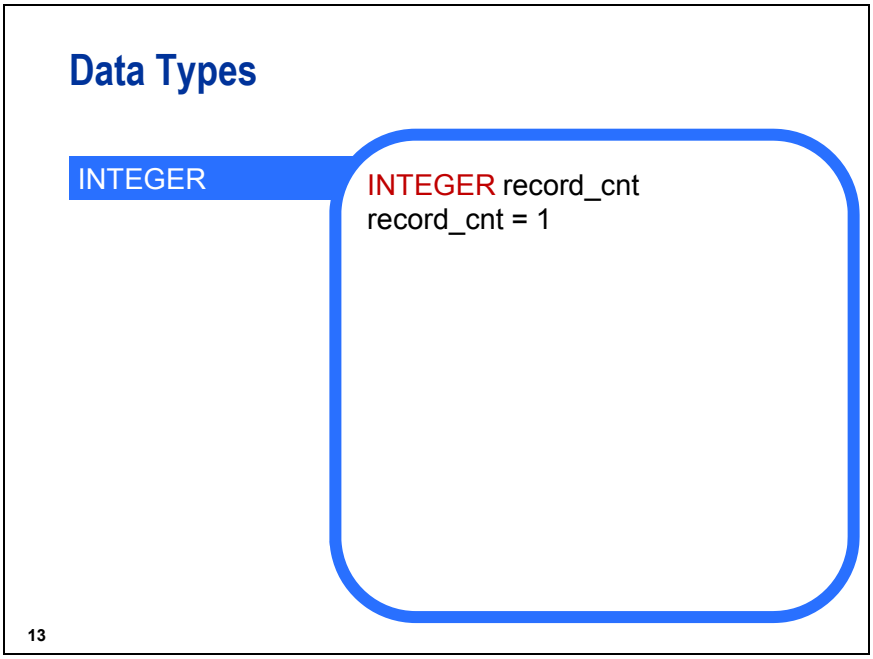
HIDDEN symbols are not output by the Expression node. That means that child nodes will not be able to use the field.

Do not confuse HIDDEN and PRIVATE:

HIDDEN symbols are visible outside the expression block.

PRIVATE symbols are not visible outside expression block.

Neither HIDDEN nor PRIVATE fields are visible to child nodes.

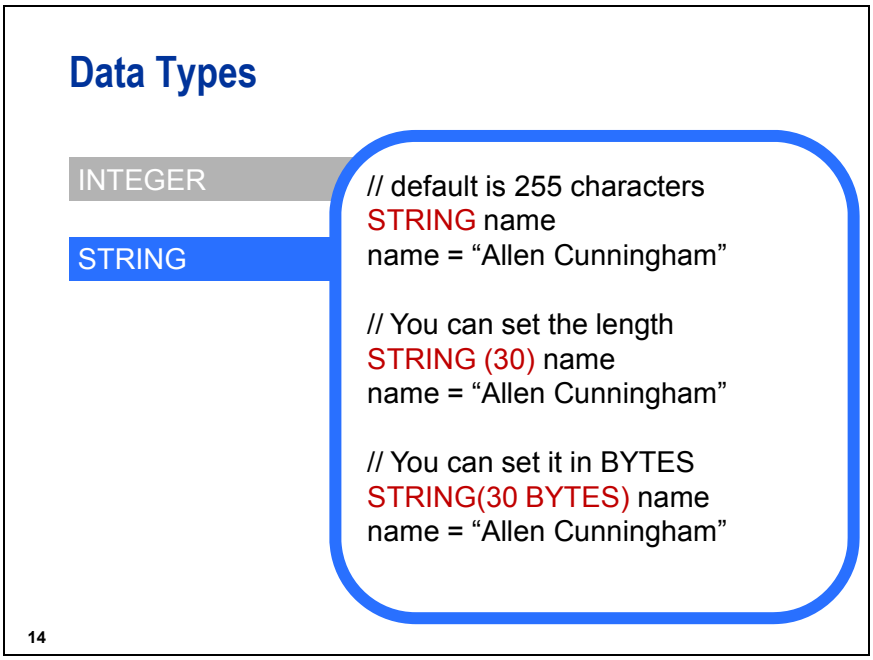


INTEGER – used for counting things, keys, etc.

This data type is not available in SAS. SAS only has number and text.

From the Expression Language Reference Guide:

Integer types are stored as 32-bit signed integers. They can have a range of values from -2^31 to 2^31-1. (-2,147,483,648 to 2,147,483,647).



// default is 255 characters

STRING name

name = “Allen Cunningham”

// You can set the length

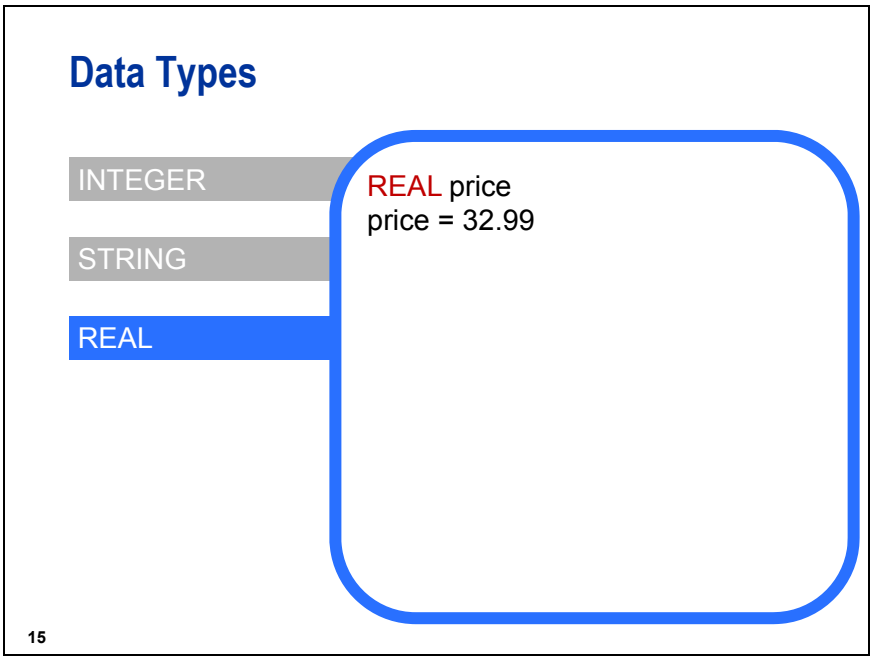
STRING (30) name

name = “Allen Cunningham”

// You can set it in BYTES

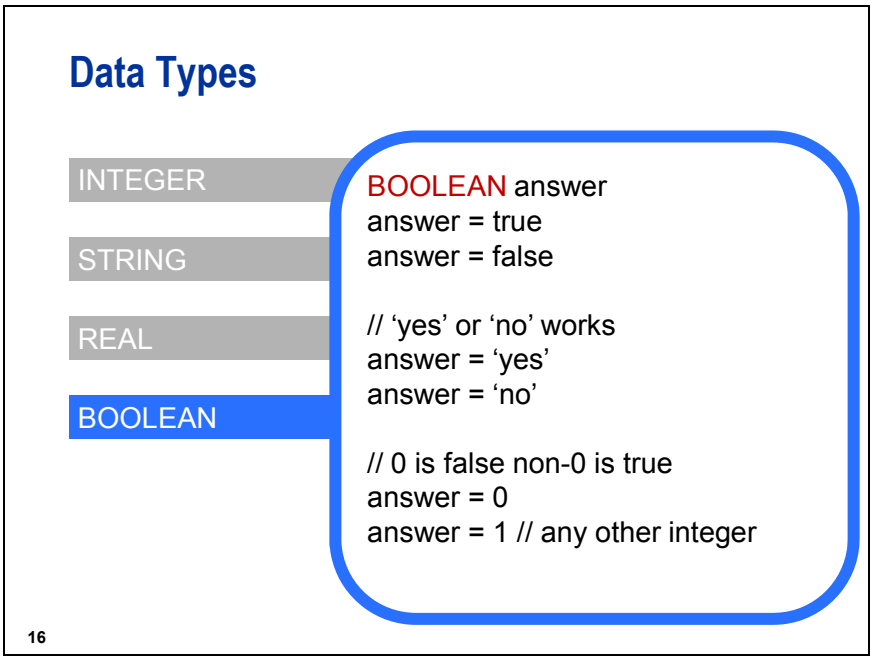
STRING(30 BYTES) name

name = “Allen Cunningham”



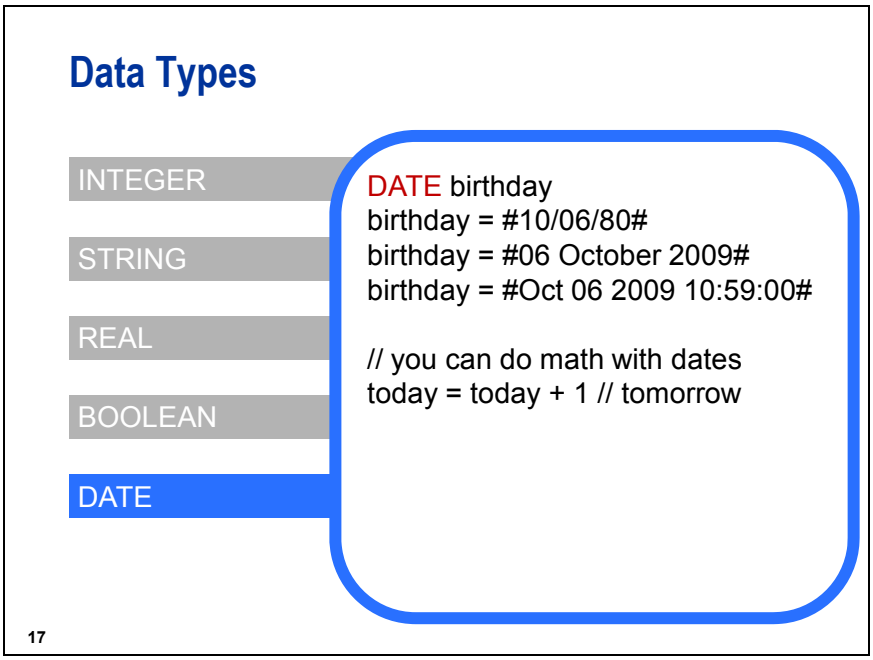
From the Expression Language Reference Guide:

Real types are stored as double precision 64-bit floats. They have a precision of 15-16 digits and a range of 5.0 x 10-324 to 1.7 x 10308. Real types are based on the IEEE 754 definition.

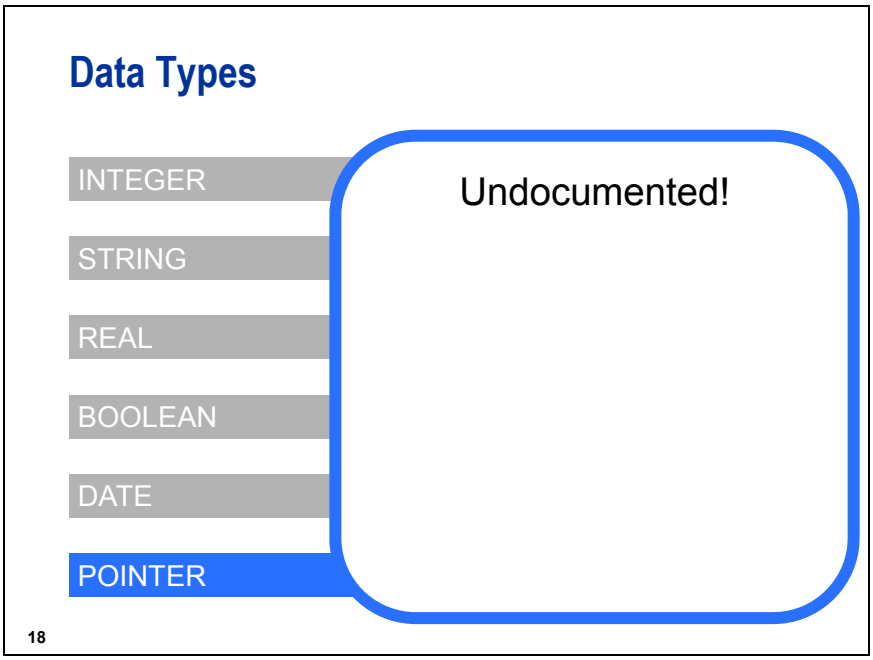


Boolean is a true/false data type. You can see that there are a couple of ways to set it. 0 or a non-zero integer. 'yes' or 'no.'

The BOOLEAN data type is used quite often in looping. We will see examples of that shortly.



The trick to setting date values is the hash mark (#). These examples should help you get started with dates. Notice that you can do arithmetic on dates.



The POINTER data type is available but it is not documented. My contacts at DataFlux don't know about this either. One is curious, and looking, but hasn't found any information on it.

Three days and no response from the DataFlux developers. I think that you can comfortably ignore this one.

Exercises

1. Match each term with the appropriate definition?

a. PUBLIC	1. Only visible in the block where it is defined
b. PRIVATE	2. Value is not reset between rows read
c. STATIC	3. Can be seen by child nodes
d. VISIBLE	4. Can be seen in other expression blocks

2. True or false, if you declare a PUBLIC variable in the Expression block can it be seen in the Post-processing Expression block? _____

3. True or False, if you declare a PUBLIC variable in the Post-Processing Expression block can it be seen in the Pre-processing Expression block? _____

4. Create a symbol declaration statement for a symbol that can ...

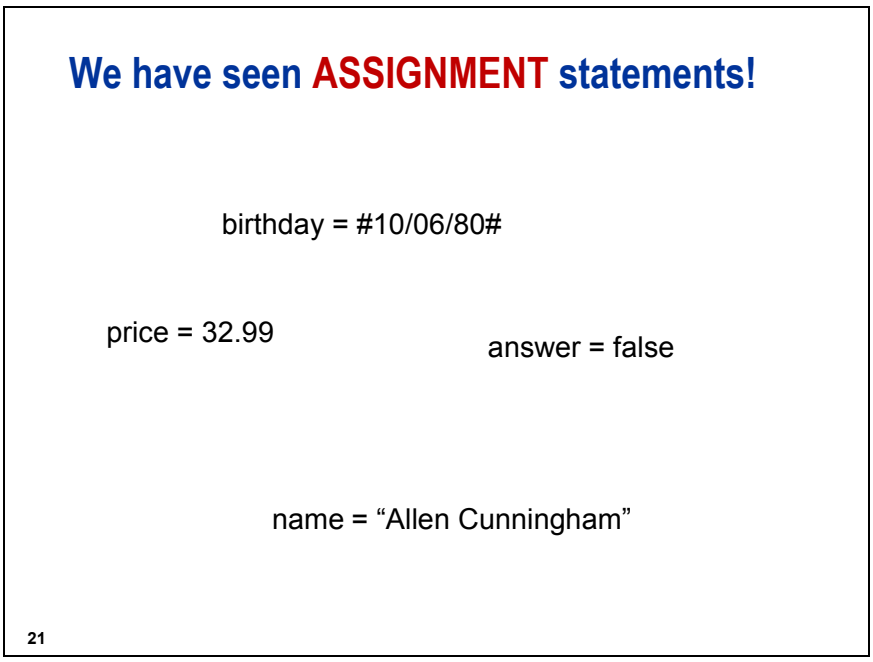
a. Hold a 50 character string that can be seen in all expression blocks

b. Hold a number like 3.14 and can't be seen by child nodes

c. Hold a date

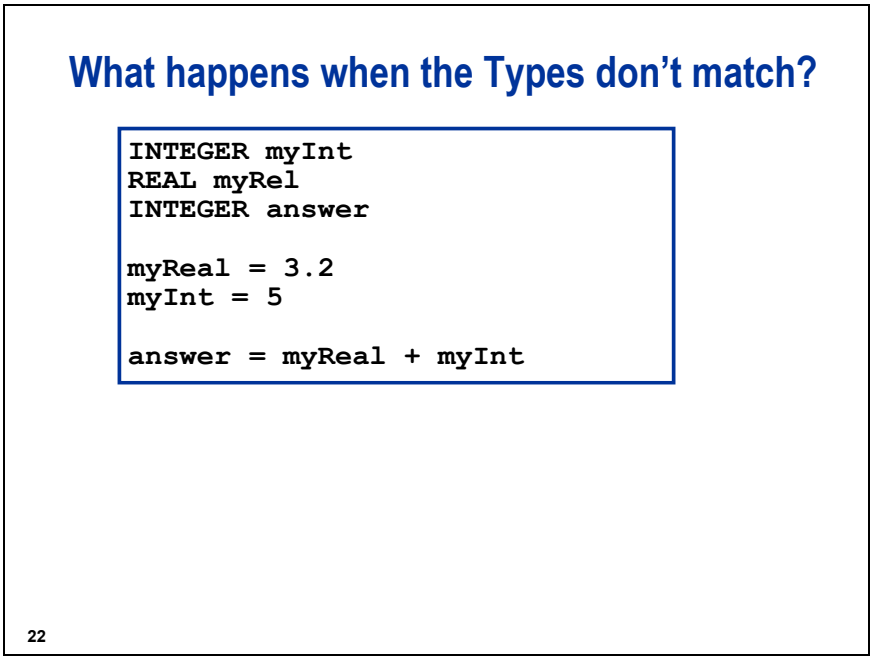
d. Hold a 30 character string named FIRST NAME

3.2 Assignment Statements



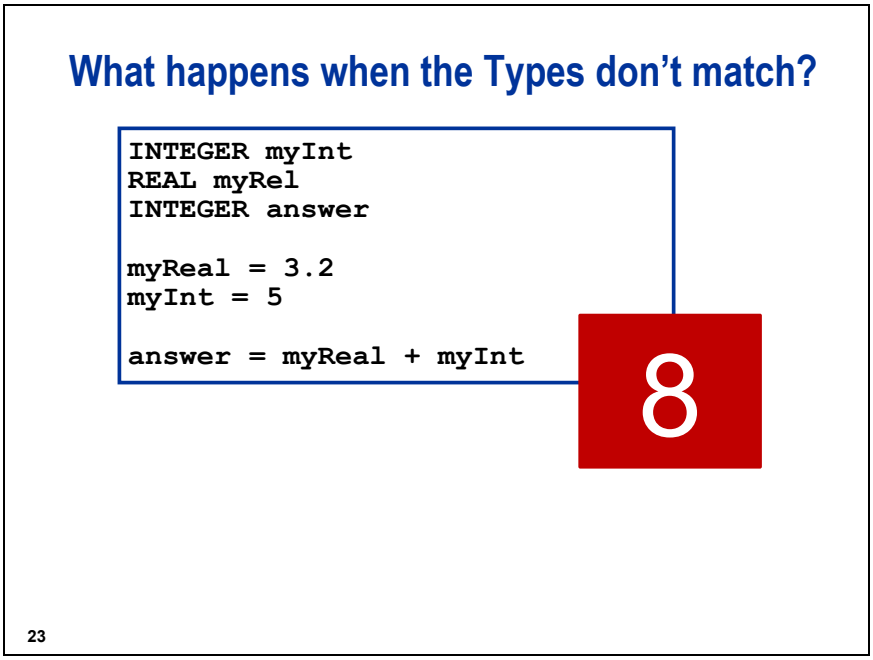
Assignment statements are used heavily in programming. Here are some of the statements that we have seen so far. These are simple. Well, the date example is pretty odd looking… More about that later.

Let’s take a look at some tricky examples.

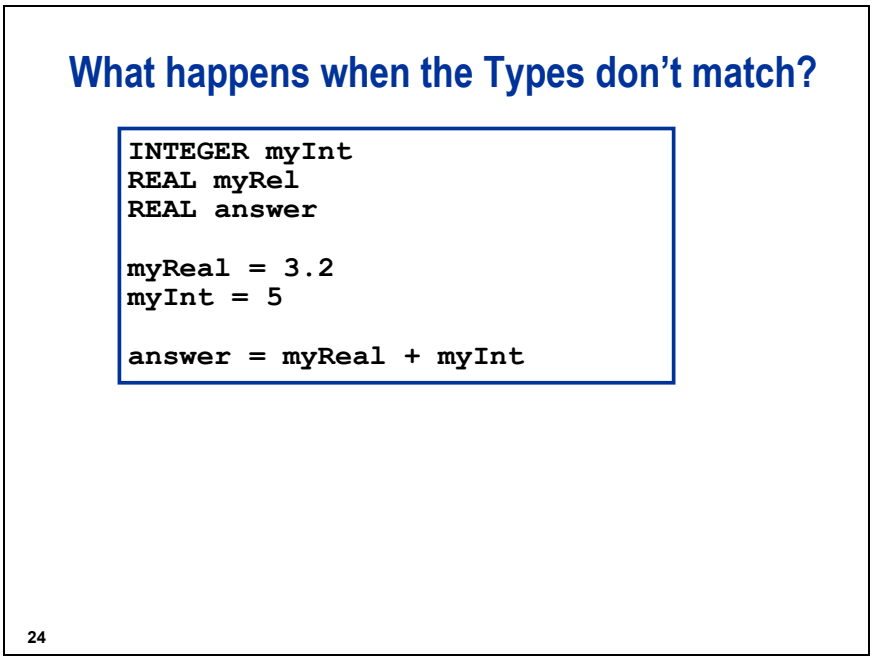


In the code we are taking a real number (3.2) and adding it to an integer (5). We are storing the value in the symbol answer (an INTEGER).

Can you guess what the value stored in the symbol “answer” will be?

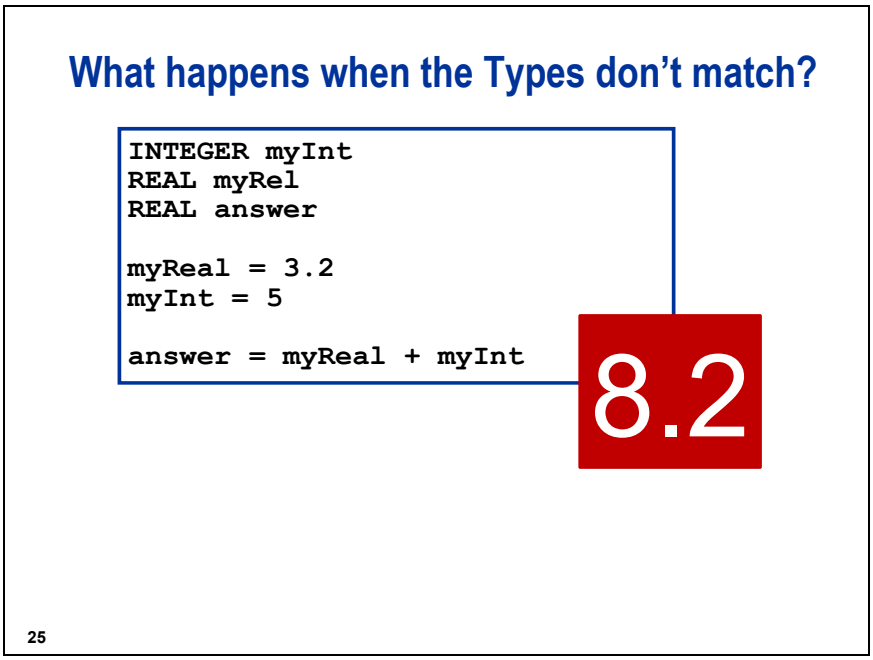


The number 8 will be stored in the symbol “answer.” Since the target is an integer, the value is truncated.



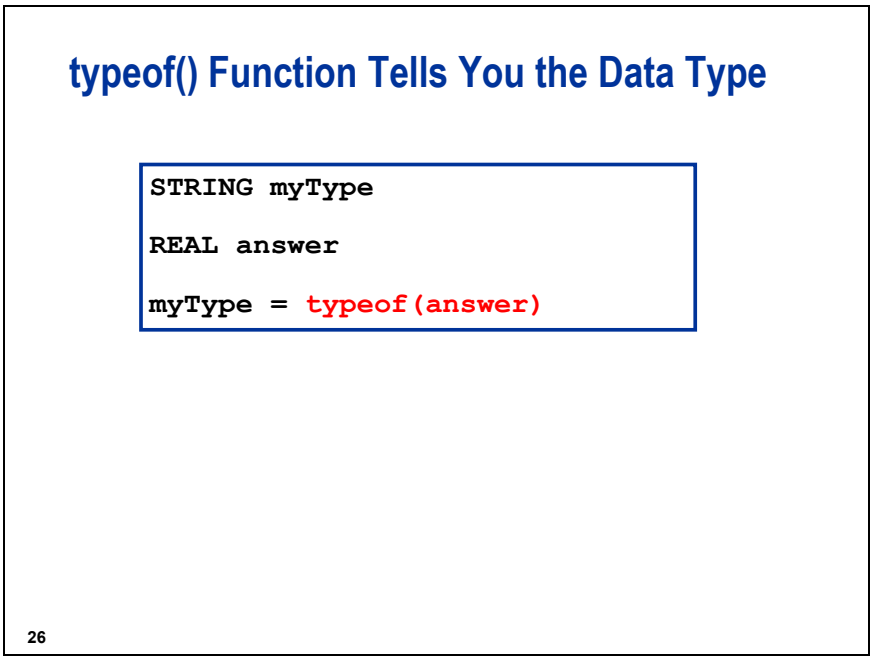
Here is another example. We are adding an integer to a real and storing the result in a real.

What is the value that will be stored in the symbol “answer?”



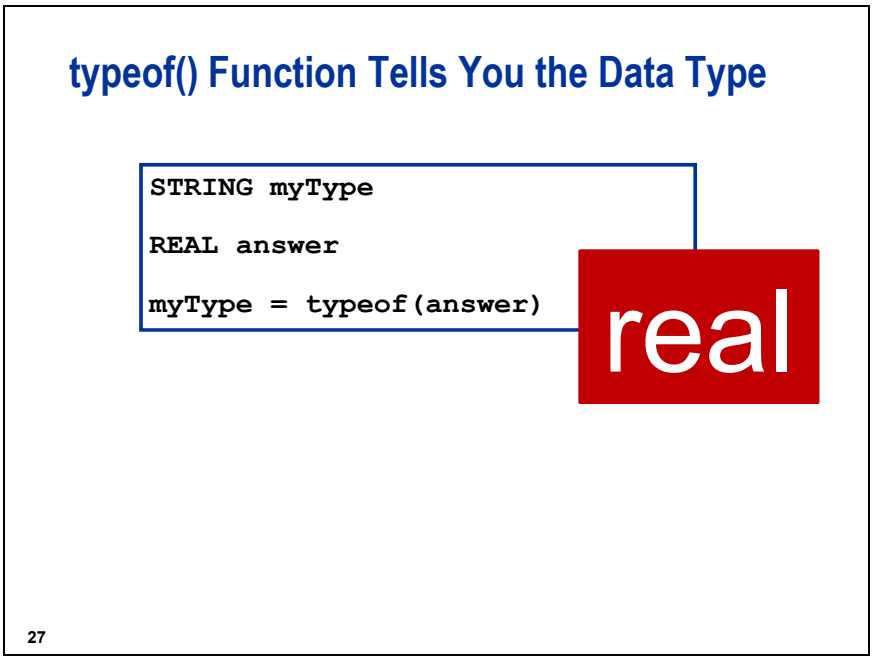
8.2! Answer is now declared as a real. There is no truncation.

This makes sense.

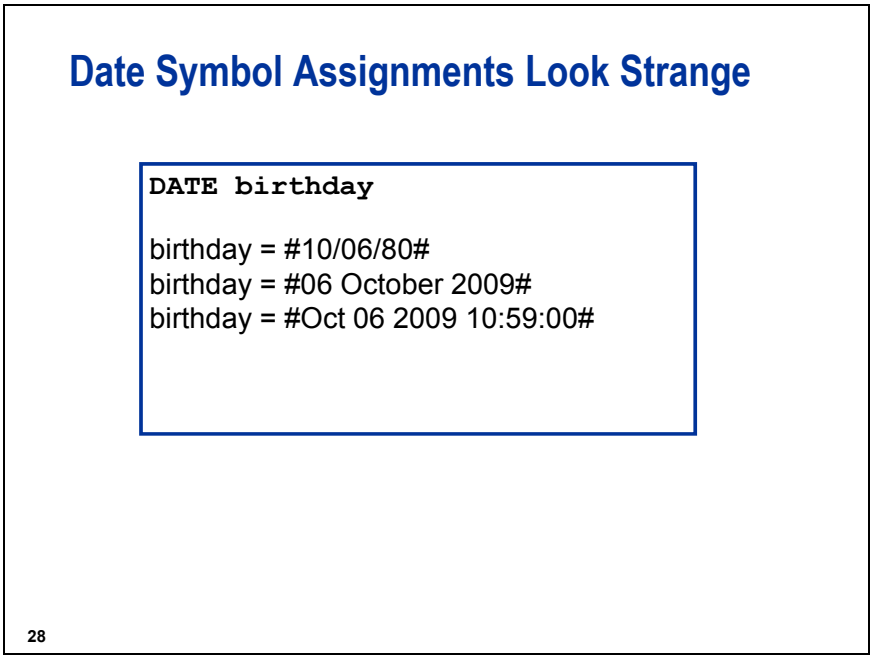


A function is a block of code that can take arguments and returns one value.

In this code we are passing a symbol name to the typeof() function. The function returns a string which states the data type of the argument. Can you guess what the function will return?

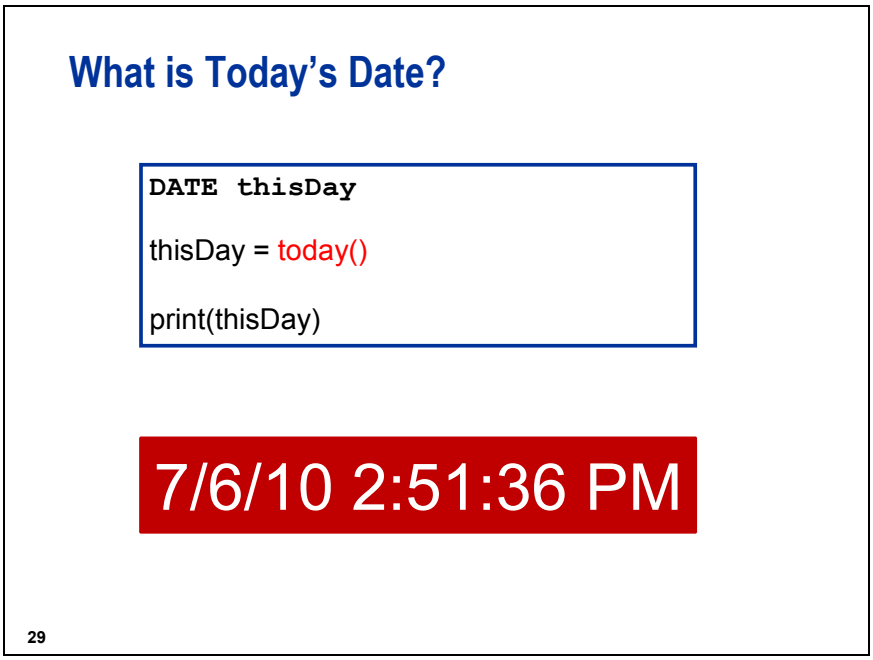


"answer" is of type real and that is what is returned.



We saw these earlier. Dates do look strange. In DataFlux dates must begin and end with a hash sign(#).

There are functions which return dates.



If your program needs to know today's date then you can use the today() function. It returns the date and time.

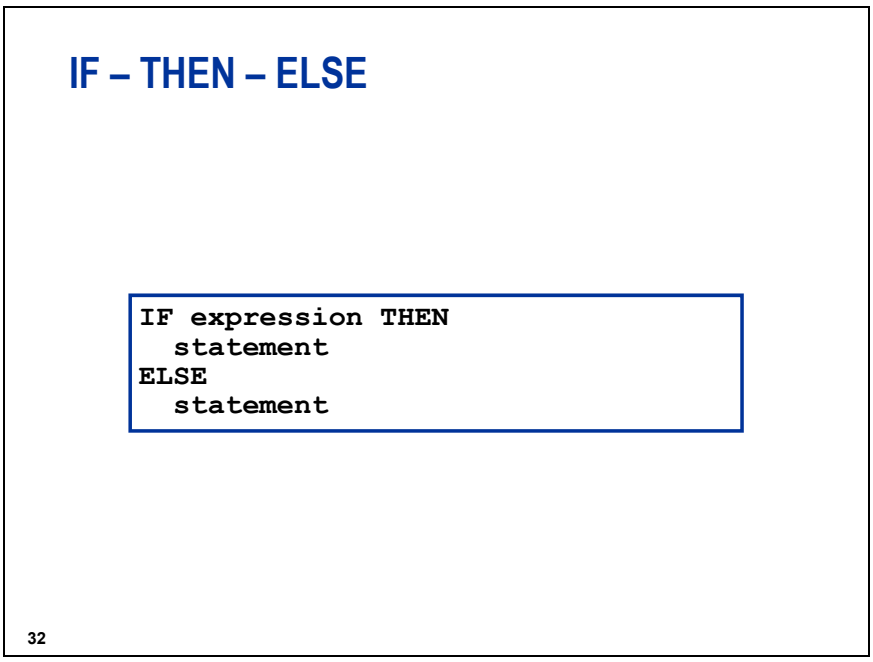
Exercises

5. Can you add a string to a number? Write a program that does this. Store the answer in a symbol of type REAL and print it to the Log using the print() function.

6. Create a program that adds 1.1 to “This is a string”. Store the answer in a symbol of type REAL. What happens?

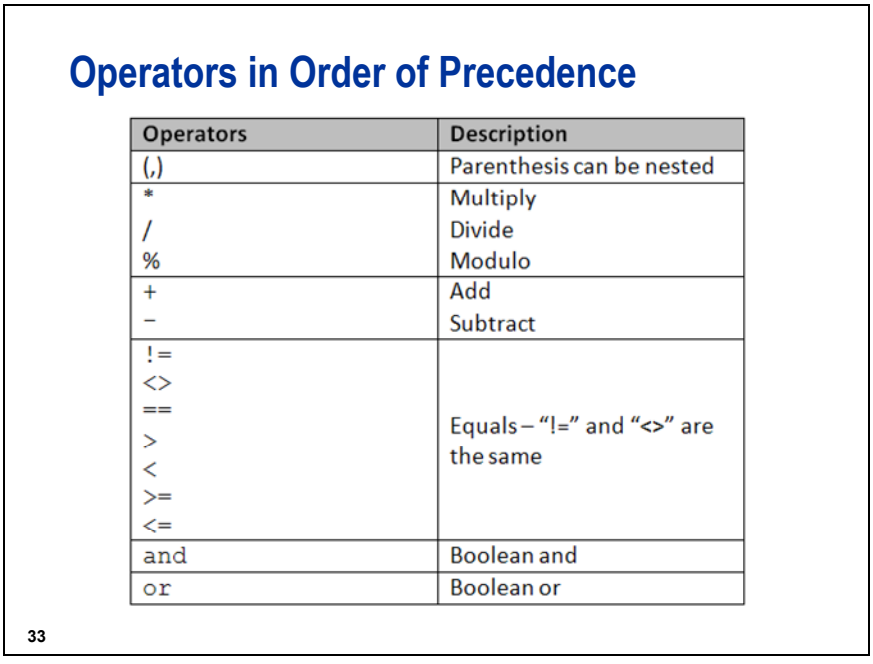
7. What does the typeof() function do?

3.3 IF – THEN – ELSE



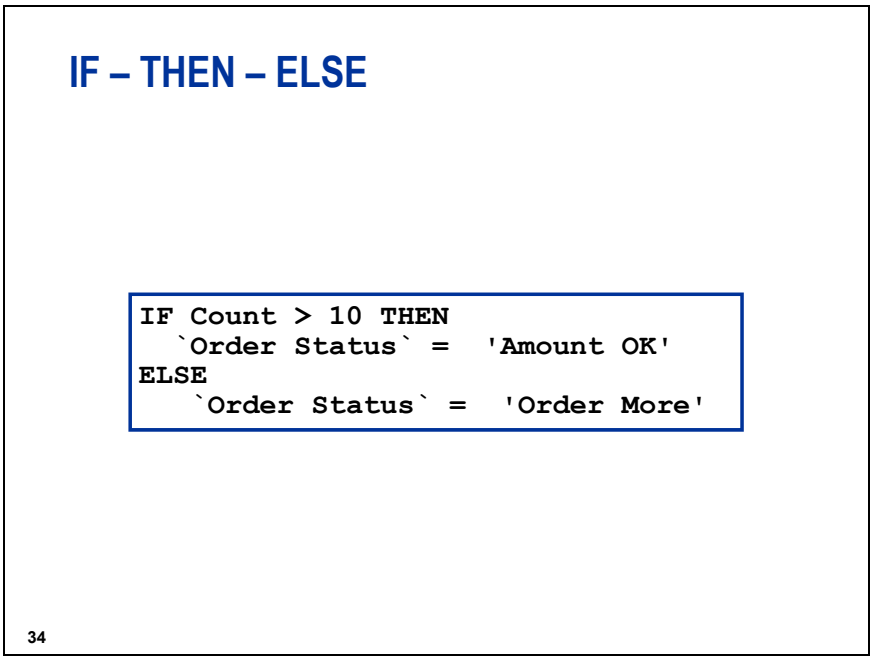
The IF – THEN – ELSE control structure is very straight forward. If you have programmed in other languages it will make perfect sense to you.

In this example an expression is a arithmetic or logic construct that evaluates to either TRUE or FALSE.



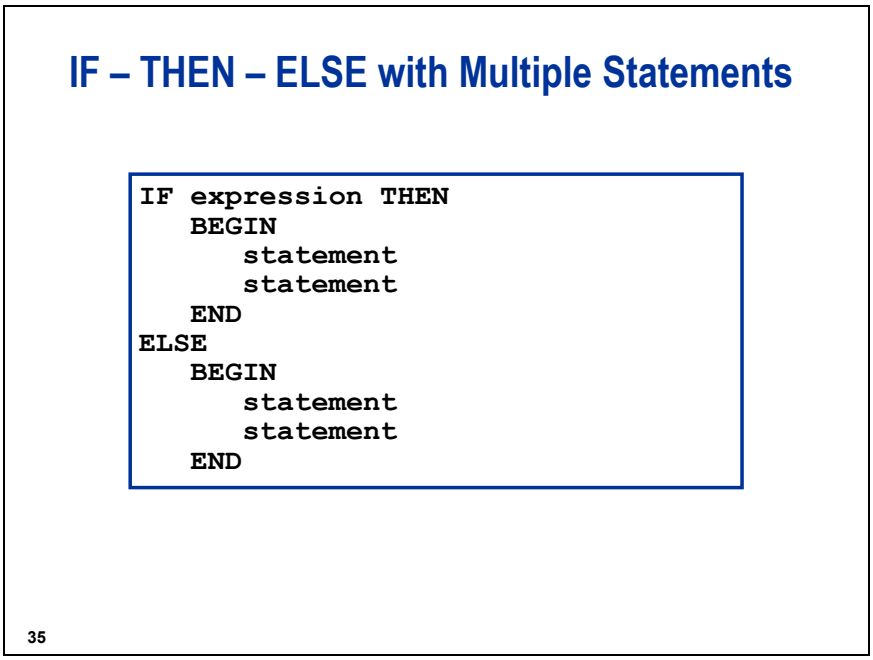
Operators	Description
(,)	Parenthesis can be nested
* / %	Multiply Divide Modulo
+ -	Add Subtract
!= <> == > < >= <=	Equals – "!=" and "<>" are the same
and	Boolean and
or	Boolean or

The precedence matches other languages. For more information check-out page 10 of the DataFlux Expression Language Reference Guide.

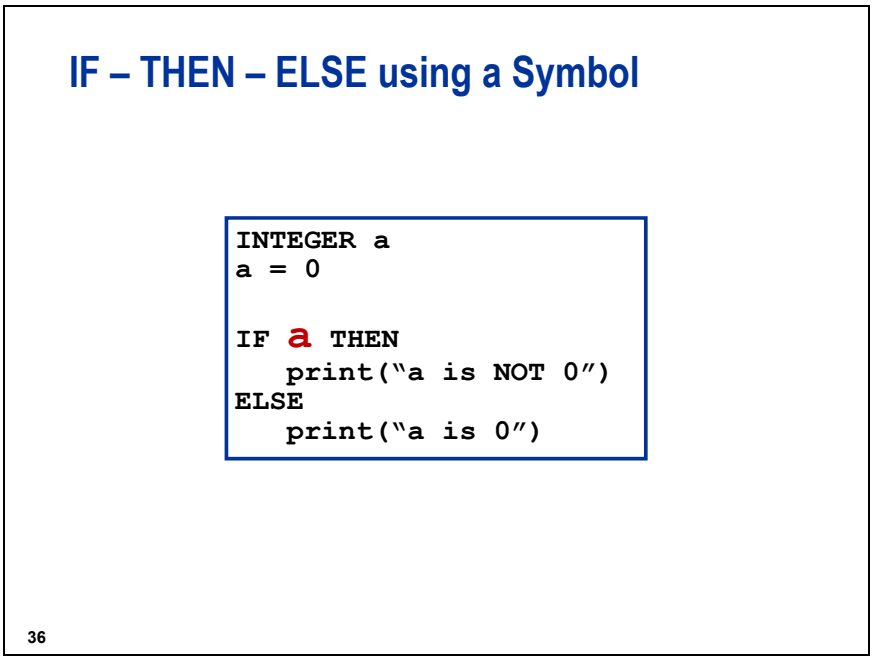


If the expression, Count > 10, is true then the `Order Status` will be set to ‘Amount OK’. If Count is less than or equal to 10 then `Order Status` is set to ‘Order More’.

Note the grave accent marks surrounding the `Order Status` symbol.

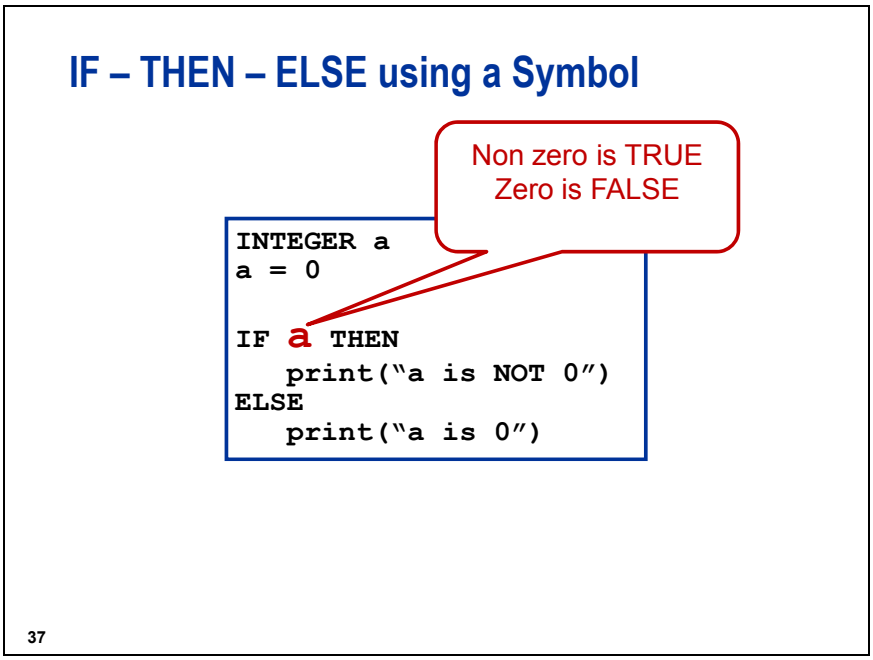


You can use BEGIN and END to include multiple statements in an IF/THEN/ELSE clause. Consistent indention makes these easier to read.



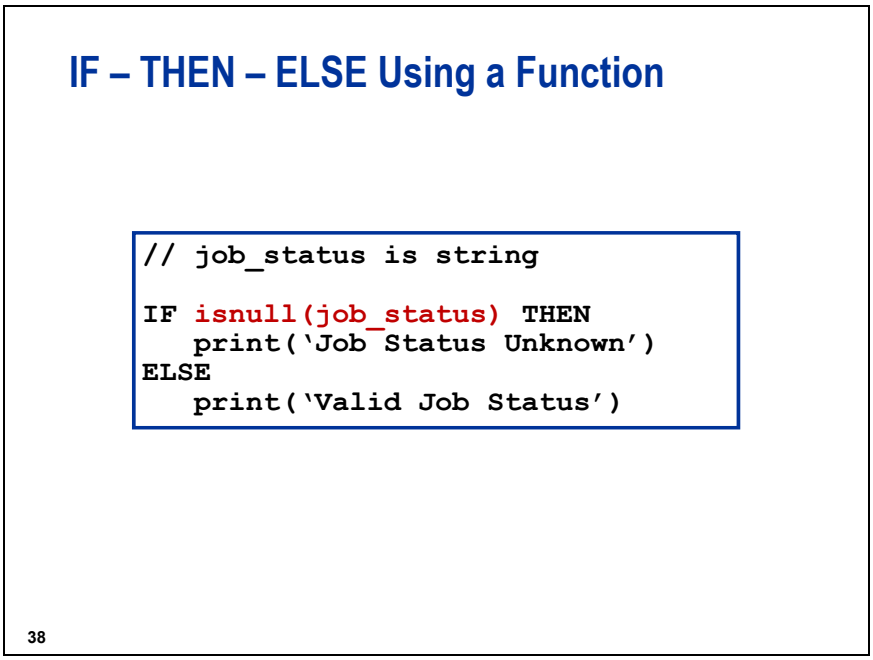
The print() function will write the included text to the Log when you preview the output. It is quite useful when testing code.

Carefully look at the expression included in this code snippet. Do you know what it does?



When a lone symbol is the expression then TRUE is a non-zero value. FALSE is zero. You may need to experiment with this if you have never seen this construct before.

When using this type of IF statement it is important to provide meaningful names to variables.



This isn't really a great example but you get the point.

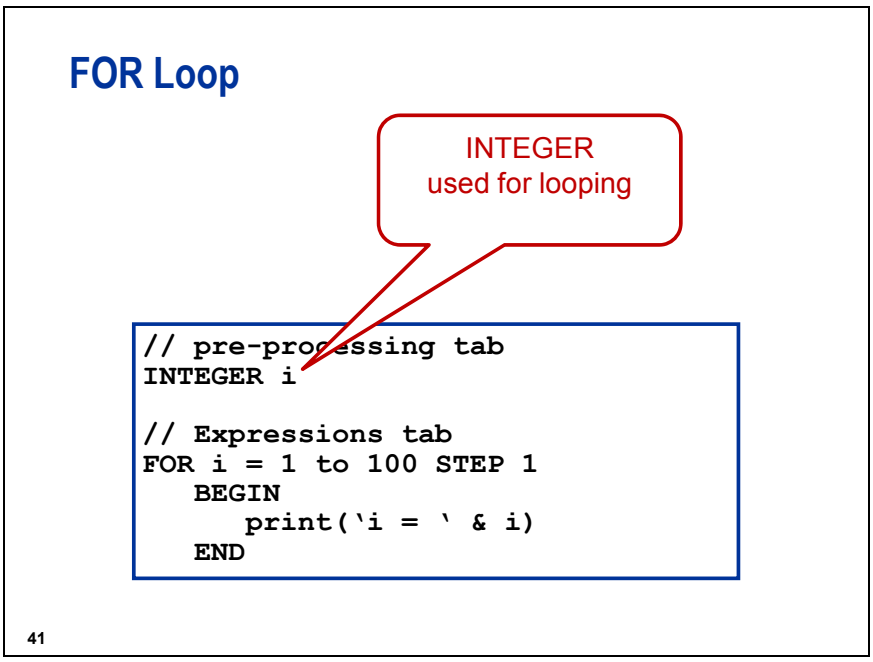
Oh, why isn't it a great example? This will count an empty string as a valid job_status. Remember, to determine the difference between an empty string and a NULL you must use expression language.

Exercises

8. Open the EEL Chapter 3 Sample 01 Architect job and debug the problem.

9. Open the EEL Chapter 3 Sample 02 Architect job and debug the problem.

3.4 Looping

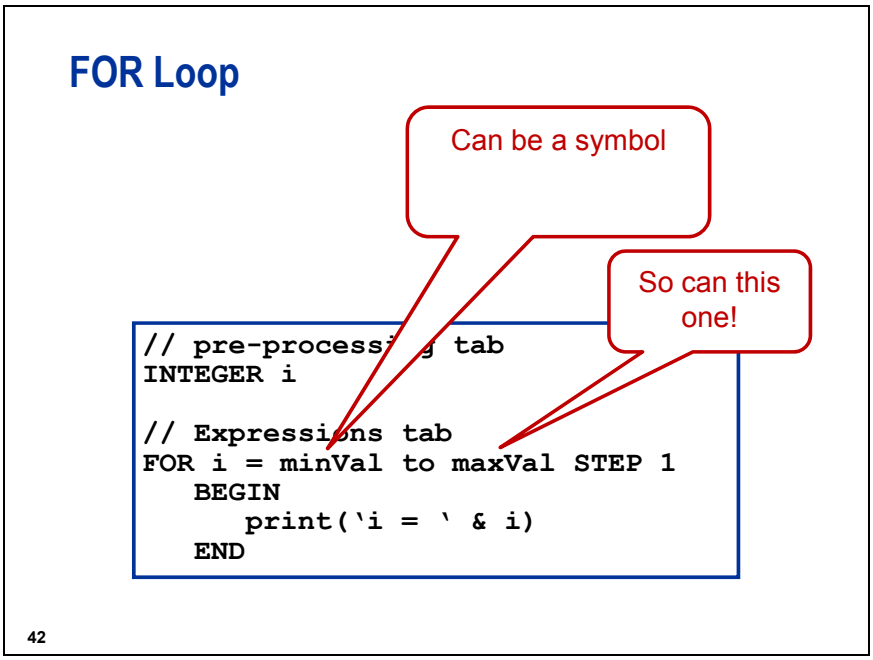


FOR loops are great for performing a set of actions a specific number of times. An integer value is used for this.

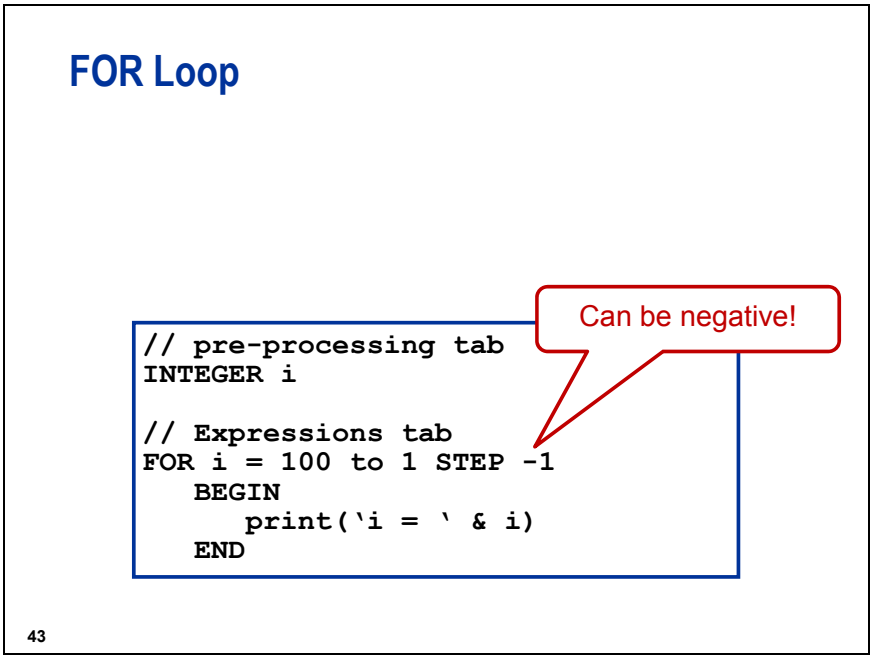
In its basic form the FOR loop starts at 1 (it doesn't have to) and continues until some it surpasses some value. In this example the upper limit is 100. The lower limit does not have to be 1 it can be any integer.

Take a look at the STEP value. In this example it is 1. That means that it is going to count 1, 2, 3, 4, … 100. When i == 101 then the statements are not executed and the loop stop.

Having the looping stop is a great idea. Infinite loops tend to cause your programs to fail.



The lower and upper bounds of the loop do not have to be simple numbers. You can put symbols in there. You can set the values programmatically. Make sure that they are set properly. For example: if you have a positive value specified for step then minVal needs to be smaller than maxVal.



You do not have to count up. You can count down. Notice that we are stepping by -1. This means that you need to make the lower bounds number larger than the upper bounds. That sounds unusual but you get the picture.

WHILE Loop

```
// pre-processing tab
BOOLEAN continue_flag
INTEGER x

// Expressions tab
continue_flag = true
x = 0

WHILE continue_flag
   BEGIN
      x = x + 1
      IF x == 10 THEN
         continue_flag = false
   END
```

44

The WHILE loop is also available in expression language. It tends to be a little more complicated than the FOR loop is. Let's take a look at the specifics.

WHILE Loop

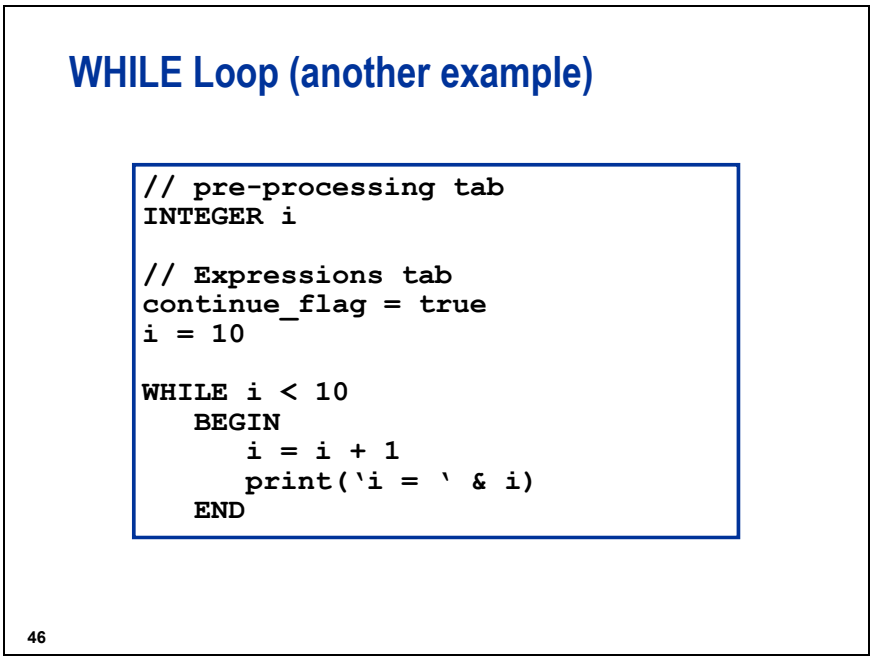
```
// pre-processing tab
BOOLEAN continue_flag
INTEGER x

// Expressions tab
continue_flag = true
x = 0

WHILE continue_flag
   BEGIN
      x = x + 1
      IF x == 10 THEN
         continue_flag = false
   END
```

While continue_flag is true the looping continues

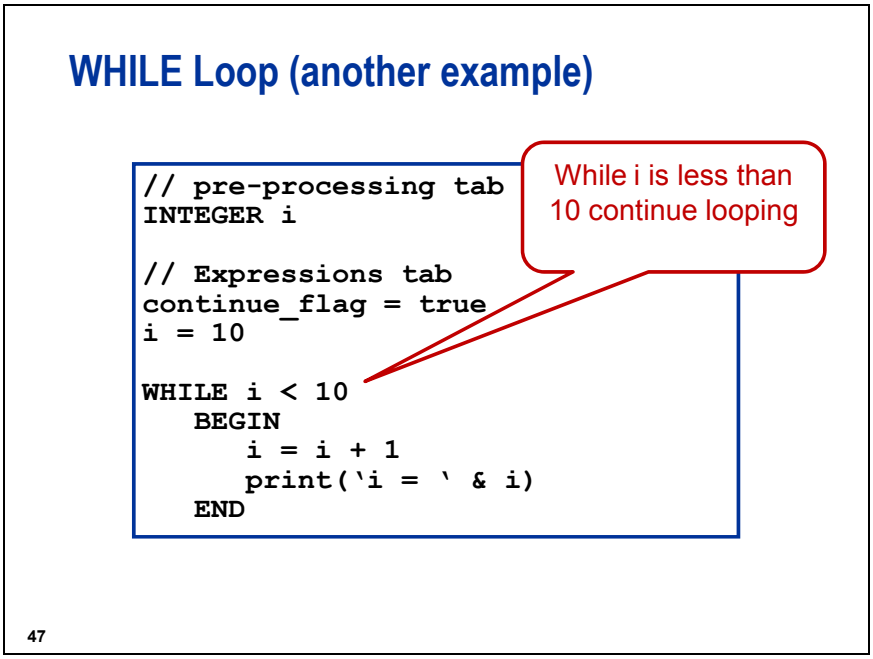
45

The WHILE loop executes until the expression, or symbol, included on the WHILE statement returns true. In this example we have a boolean value specified on the WHILE statement. While the value is true the looping continues.

You don't have to use a boolean value on the WHILE statement.



In this example we have the looping based on an expression: i < 10 .



While i is less than 10 the looping continues. Notice that we are keeping count inside the loop.

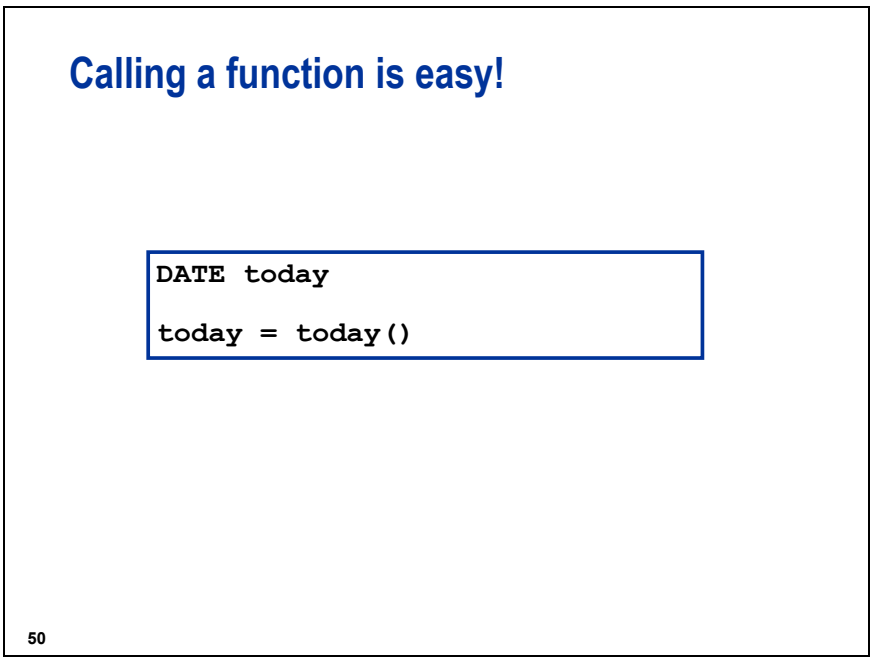
Exercises

10. Open the EEL Chapter 3 Sample 03 Architect job and debug the problem.

11. Create a WHILE loop that uses a Boolean variable to determine when looping should end. The loop should execute 10 times. Use the PRINT() function to track the loop count.

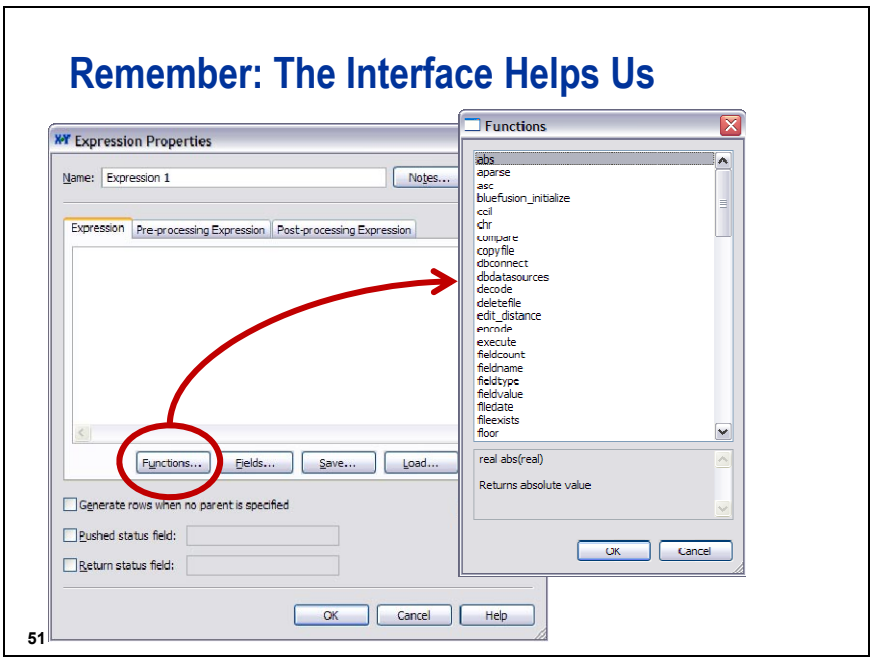
12. Create a FOR loop that starts at 10 and counts down, by 2, to 0. Use the print() function to display the count.

3.5 Functions

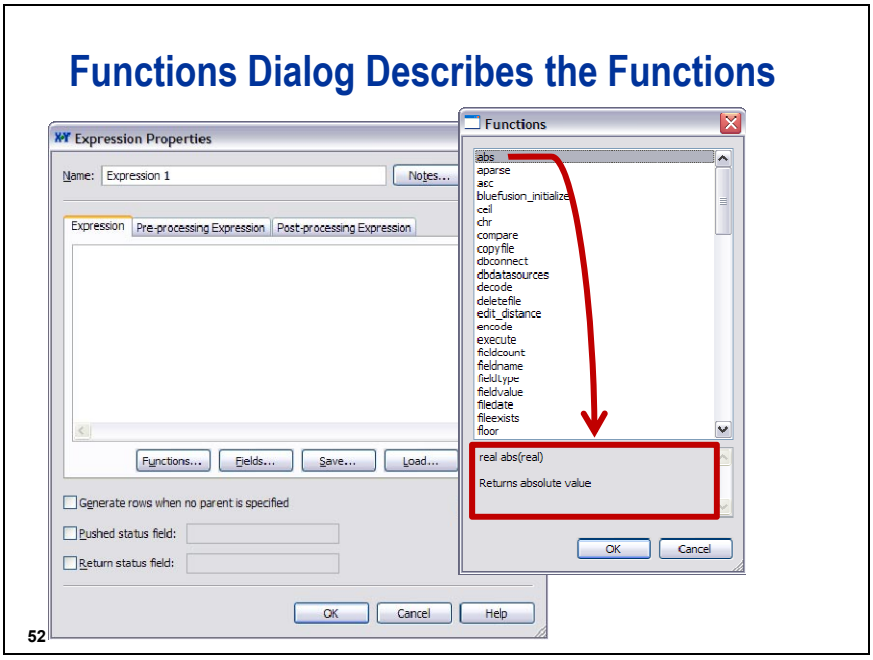


This is interesting because the function and the symbol have the same name. I am not sure that I would suggest doing this but it is possible.

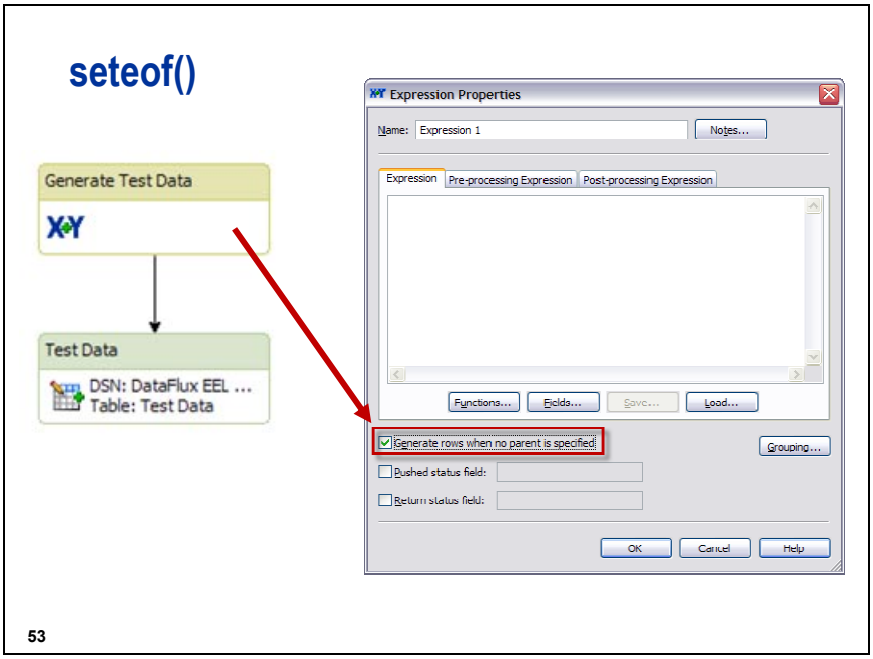
The thing to remember about functions is that they return a value of a specific type. They can take arguments but they don't have to.



We are not going to cover all the available functions here. There isn't time to do that. We will cover some of the more important ones. Alas, we don't really have time to cover all the important ones.



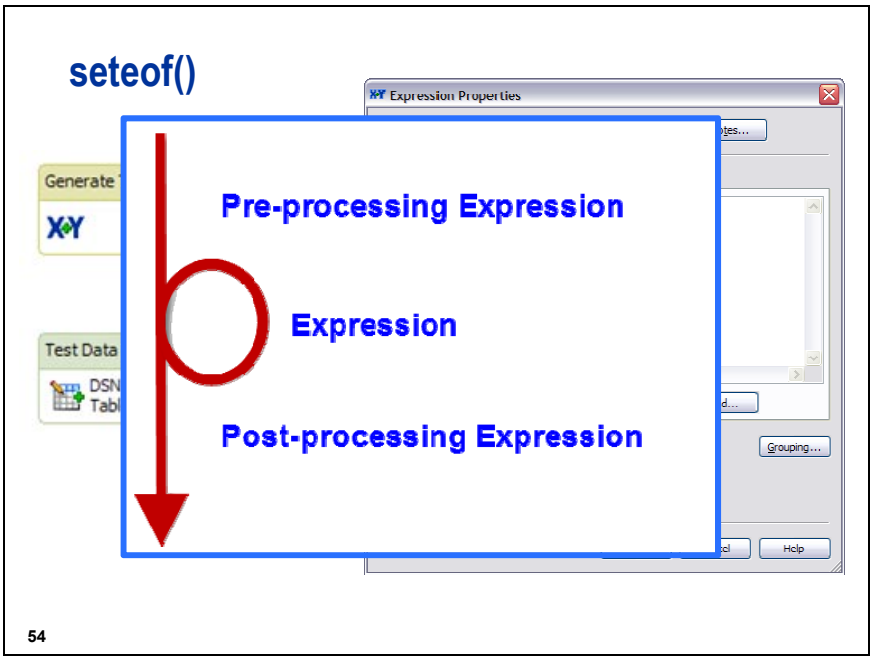
Highlighting a function displays basic information about the function: arguments, return type and a very brief description of what it does.



When you use the Expression node to create data ("Generate rows when no parent is specified") you need to have a way to stop it. That is where the seteof() function comes in.

This function signals to the Expression node that an end of file condition has been encountered; there is no more data to read. This is important! Without this function you would have an infinite loop situation. That isn't good.

Let's take a look at creating data using the Expression node.



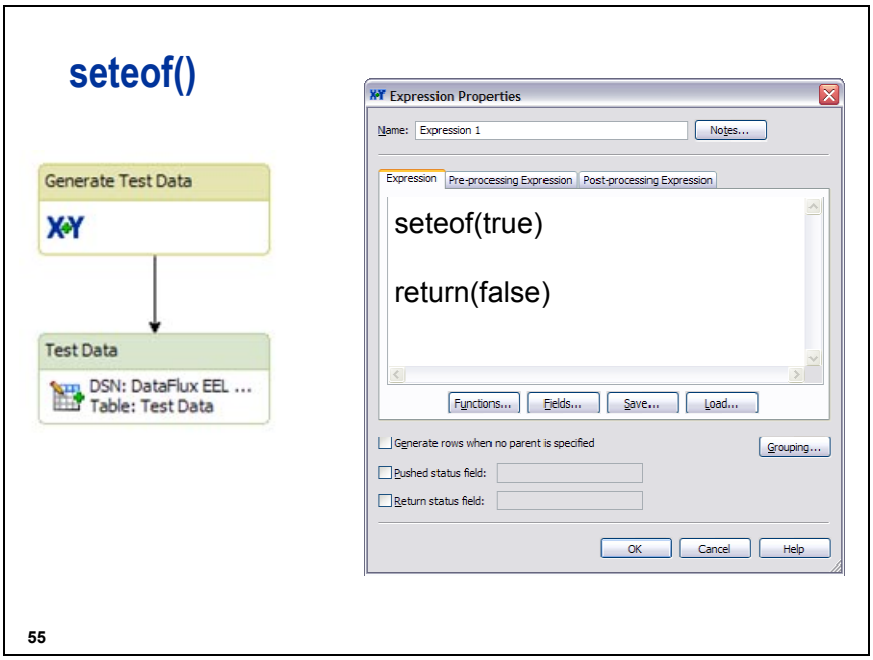
Now is a great time for a reminder of how this dialog works.

The Pre-processing expression is executed before input records are read. You may decide to declare variables and assign initial values to them here.

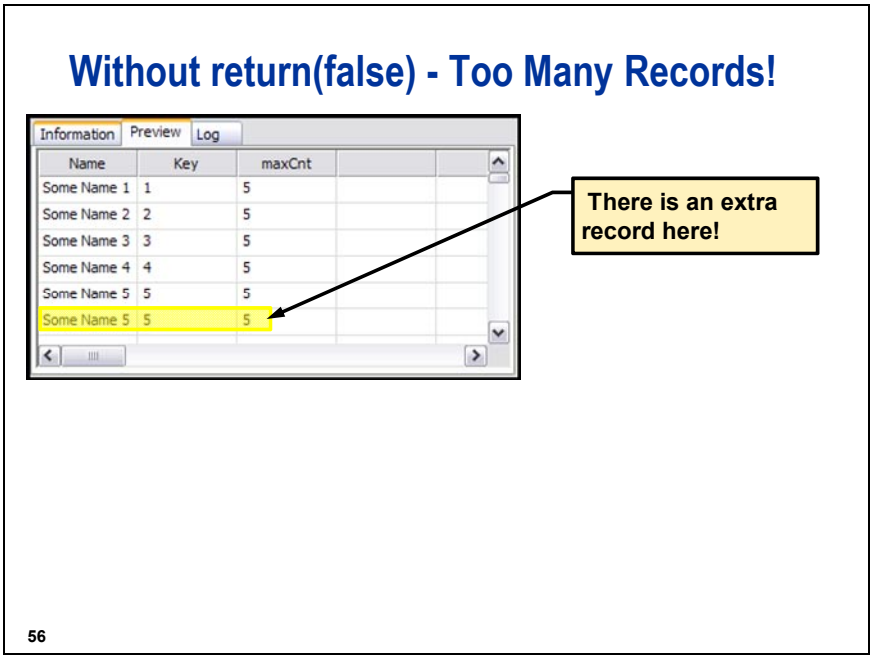
The Expression is where the action usually takes place. It is here that the input rows are looped through. If you selected "Generate rows when no parent is specified" then there is nothing to loop through. There is a sneaky little trap here. Even though there is no data you still have to stop processing. That is where the seteof() function comes in.

Finally, the Post-processing expression is executed.

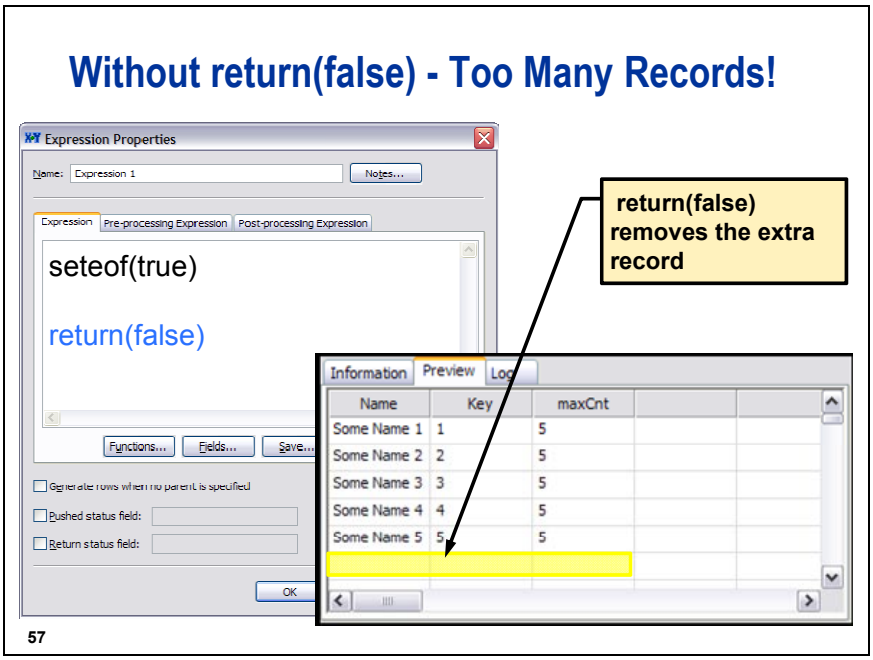
Now, back to generating data.



We know why the seteof(true) function is used here. What about the return() function, why is it there?



We know why the seteof(true) function is used here. What about the return() function, why is it there?



The return() function tells the Expression node whether or not to return a value. In this case, we don't want the node to return the record, we did it with the pushrow() function, since it causes the last row to be duplicated.

Exercises

13. What do these functions do and what data type do they return? It is best if you write code and experiment with them. Be sure to check the DataFlux Expression Engine Language Reference Guide along with other sources.

isalpha - _____

isblank - _____

isnull - _____

isnumber - _____

left - _____

len - _____

locale - _____

match_string - _____

mid - _____

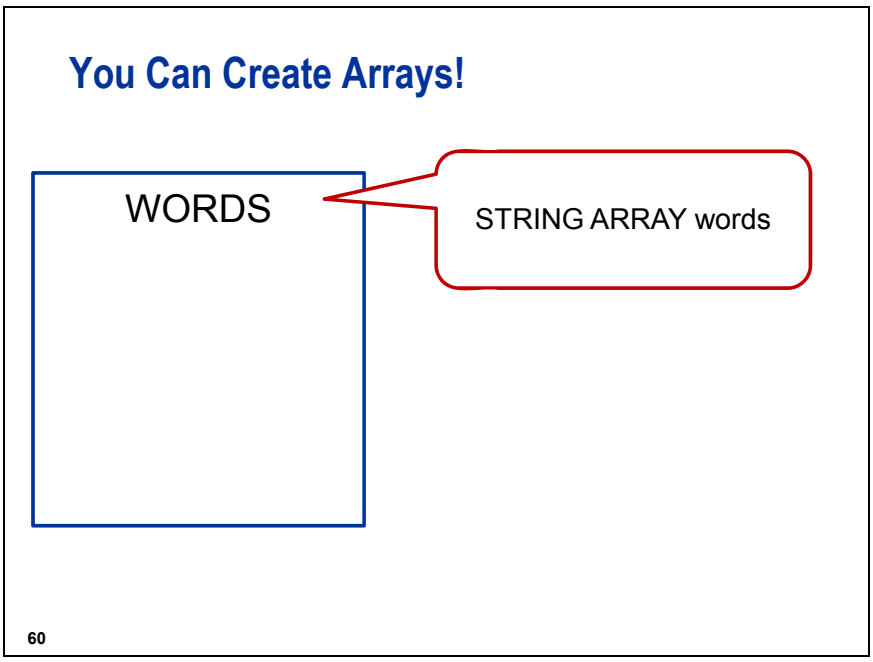
seteof - _____

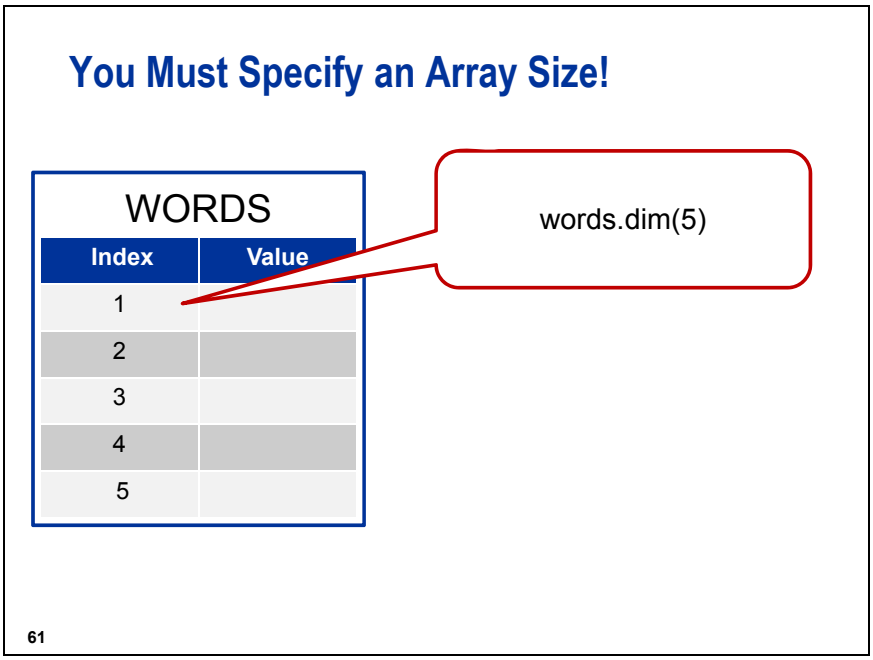
upper - _____

14. Which function allows you to get the value of a DataFlux macro variable so that you can use it in your expression language code?

15. Which function allows you to set a DataFlux macro variable in expression language code?

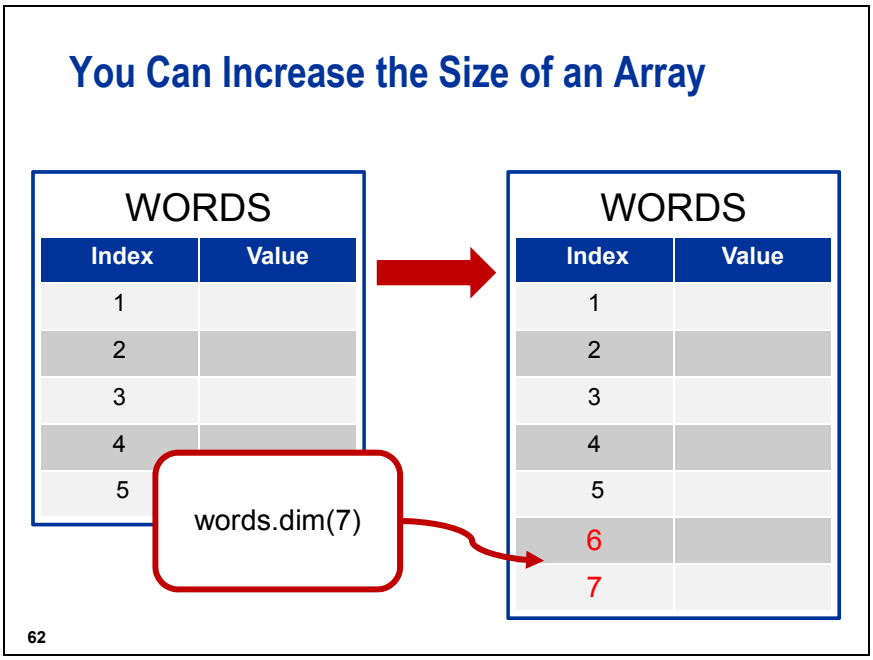
3.6 Arrays





Arrays, you will know when you need them. We won't spend much time covering Arrays. Just show the syntax and the functions (methods) that are required to manipulate them.

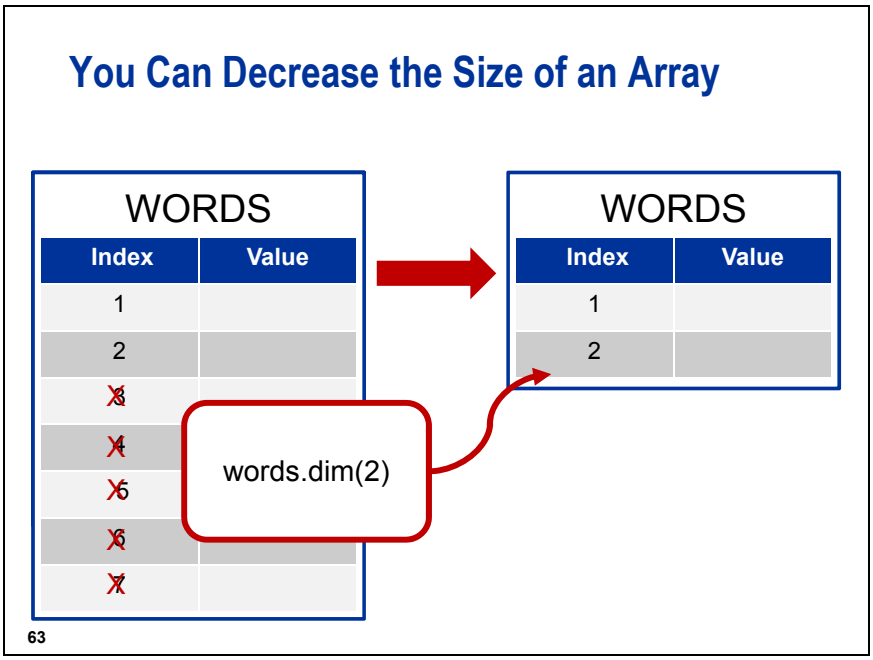
I may need to find some reason that arrays are used in EEL.



You can increase the size of an existing array by re-executing the dim() method with a larger value.

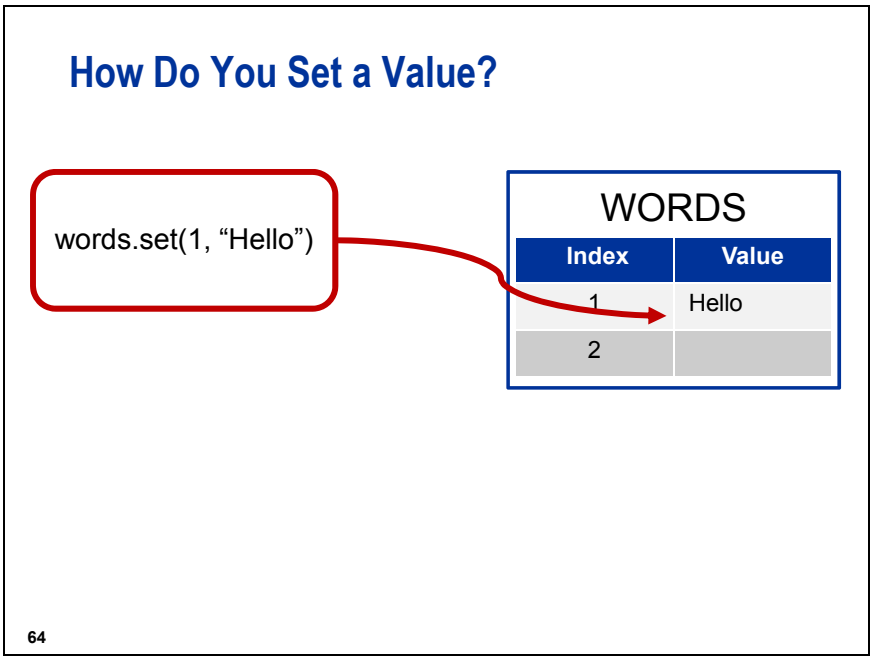
What happens to the existing data if you do this?

What is the value assigned to the new array elements?



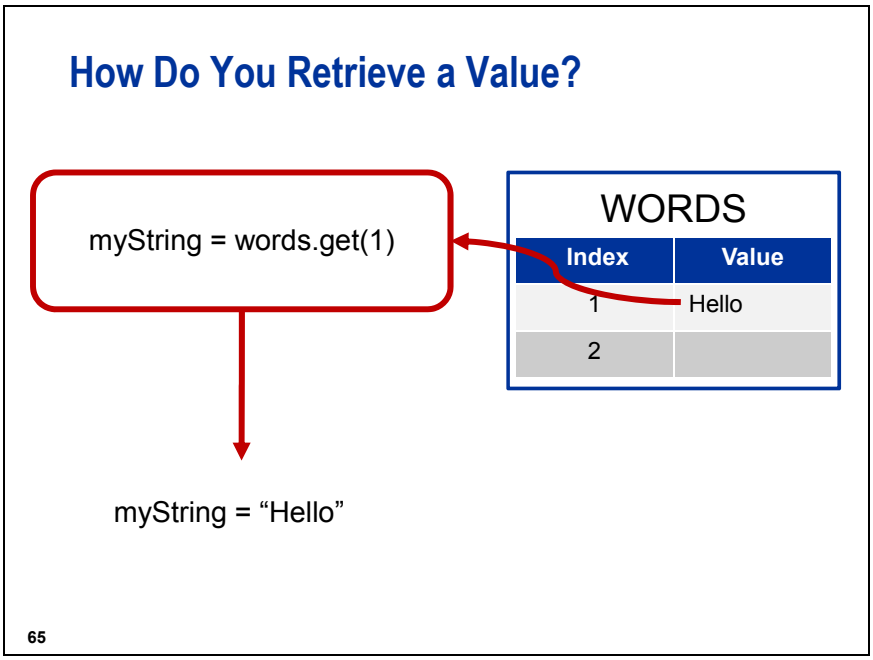
Decreasing the size of the array works the same as increasing the size. Re-invoking the dim() method with a smaller integer value will make the array smaller.

What do you think will happen to existing data in the array?



Setting the value of an array element is easy. You use the set method to do this. You will need to pass the index value (1 in this example) and the actual array value (“Hello”).

If you specify an invalid value for the index, or an invalid element value, your program will fail.



Retrieving the value stored in an array element is easy; You use the get method (via an index value to do this). If you specify an invalid value for the index your program will fail.

Exercises

16. Create an Architect Job that reads a list of 11 names (30 characters max) and stores them in an array named Names. Use a loop to write these names to the Log.

Input Data: Place a Job Specific Data node on the page. Call this node Name List. Create a Field named Name. Make it a STRING of length 30. Add 11 values to the node.

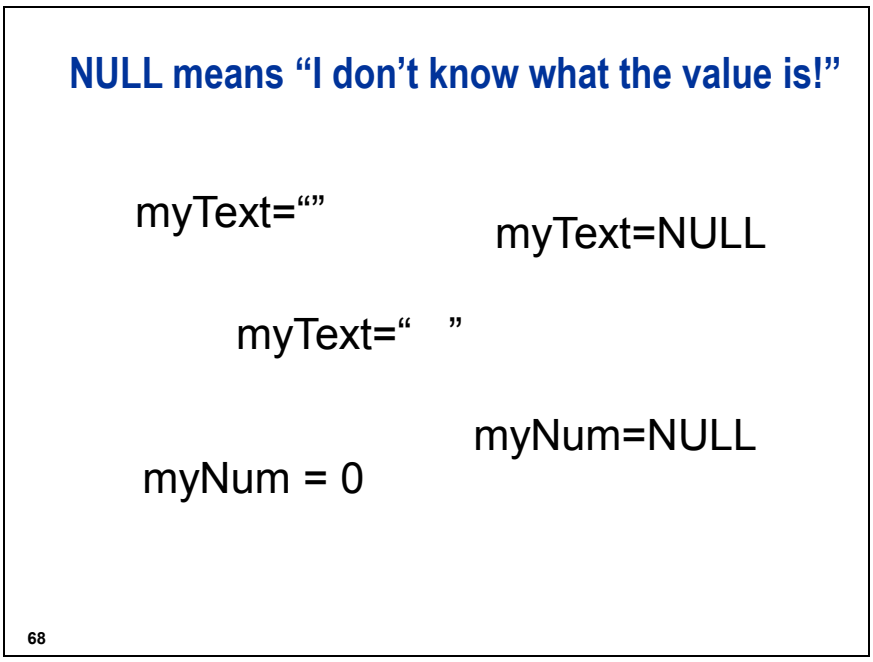
The Name field is the only field that should be output by the Expression node. You will need to create symbols to get the program to work. The trick is to not pass them out of the node.

Use an array named Names for your work. Size it so that it will hold your list of 11 names. Add the names to the array as they are read by your Expression node. Once all the rows have been read use the print function to write the list to the Log.

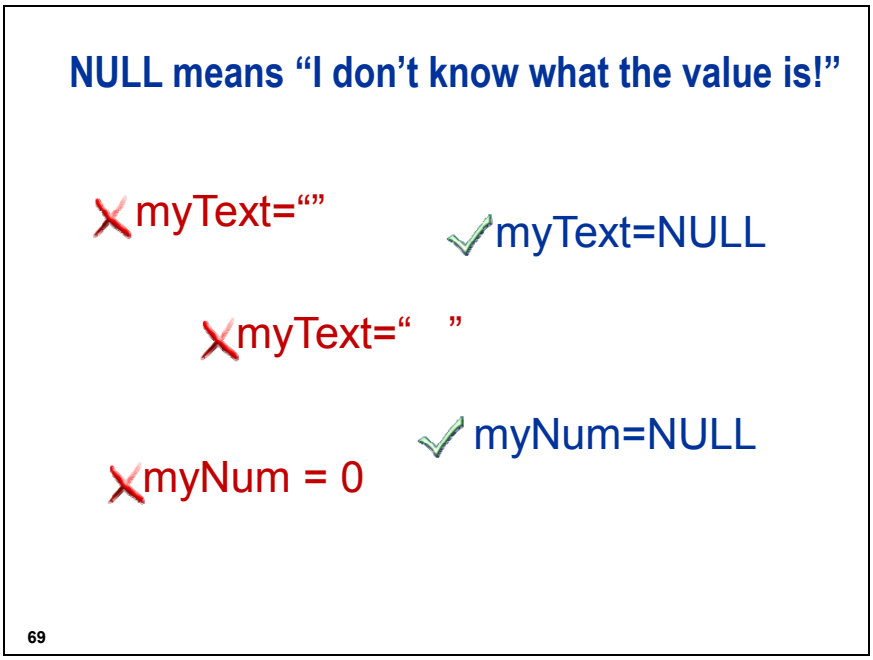
Your print function may look similar to this (the & concatenates strings):

```
print("Index --> " & index & " Value --> " & Names.get(index))
```

3.7 NULL Values



NULLs are important. NULL means that you don’t know what value a variable has.



Many people confuse empty strings with NULL values. An empty string is a string with nothing in it – think quotes with nothing separating them “” – or a string with nothing but white spaces. These are both known values.

A NULL means “I don’t know what the value is.” It comes from the relational database world.

A numeric variable can be NULL as well. Keep in mind that a value of 0 is not a NULL value.

To set a value to NULL you put the word NULL – with no quotes – on the right side of the equal sign.

How do I tell if a value is NULL?

isnull() Function

```
STRING name

name = NULL

IF isnull(name) THEN
   print("name is NULL")
ELSE
   print("name is NOT NULL")
```

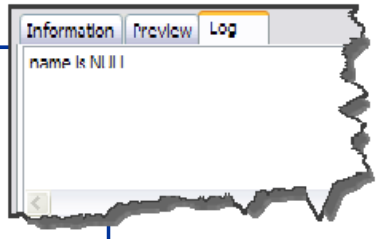
70

In expression language you must use the isnull function to determine if a value is NULL. That makes sense.

How do I tell if a value is NULL?

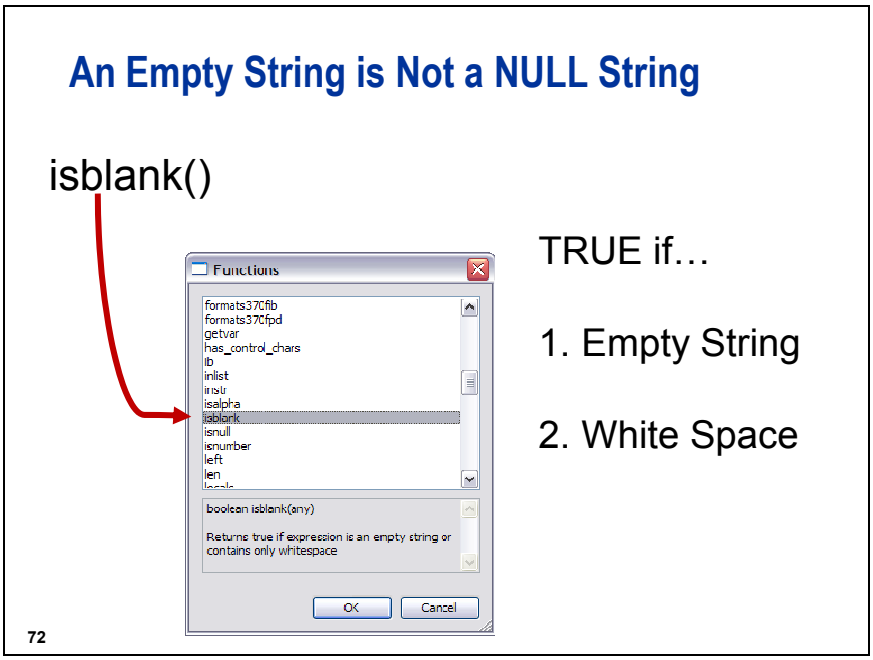
isnull() Function

```
STRING name

name = NULL

IF isnull(name) THEN
   print("name is NULL")
ELSE
   print("name is NOT NULL")
```

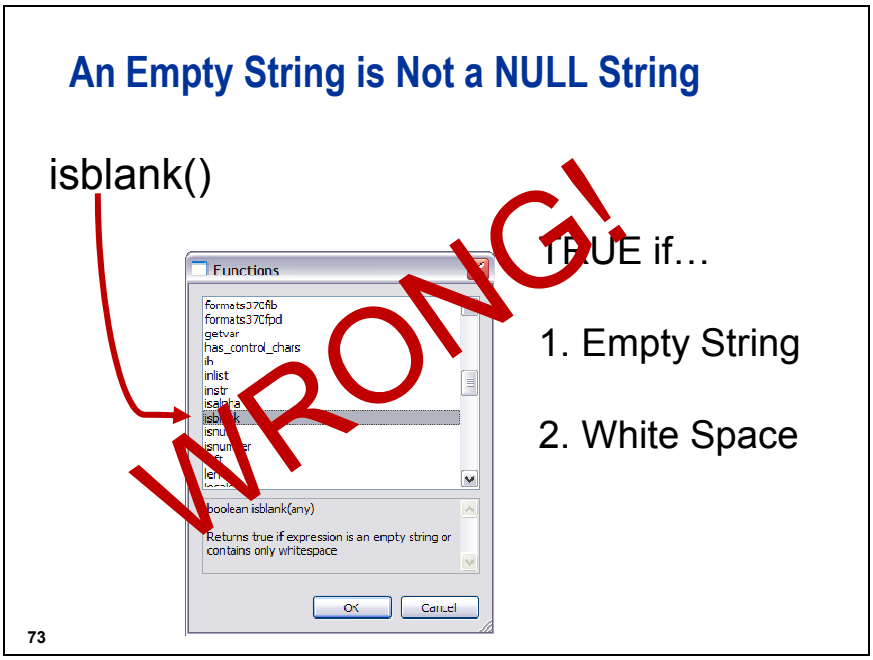


71

The output in the Log shows that the name variable is NULL.



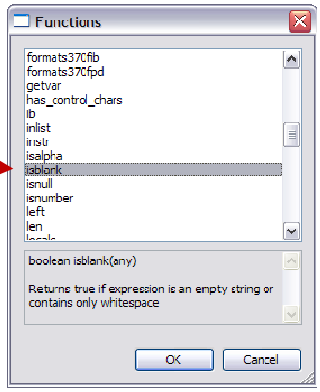
Ah, the isblank() function. You use this to determine whether a field is an empty string or contains white space. But! All is not well…



This is not quite true. Actually, it is frustratingly WRONG! There is something missing.

An Empty String is Not a NULL String

isblank()



TRUE if…

1. Empty String

2. White Space

3.NULL

74

The fact that the isblank () function returns true for NULL values complicates things somewhat. An empty string and a NULL are not the same thing. To learn whether a value is really an empty string you must use and AND condition.

Unfortunately, the dfPower Studio 8.1.1 application help is wrong. The information in the DataFlux Expression Language Reference Guide and the 8.2.1 Functions dialog is correct.

The question remains: How do we find only empty strings (including strings with only whitespace)?

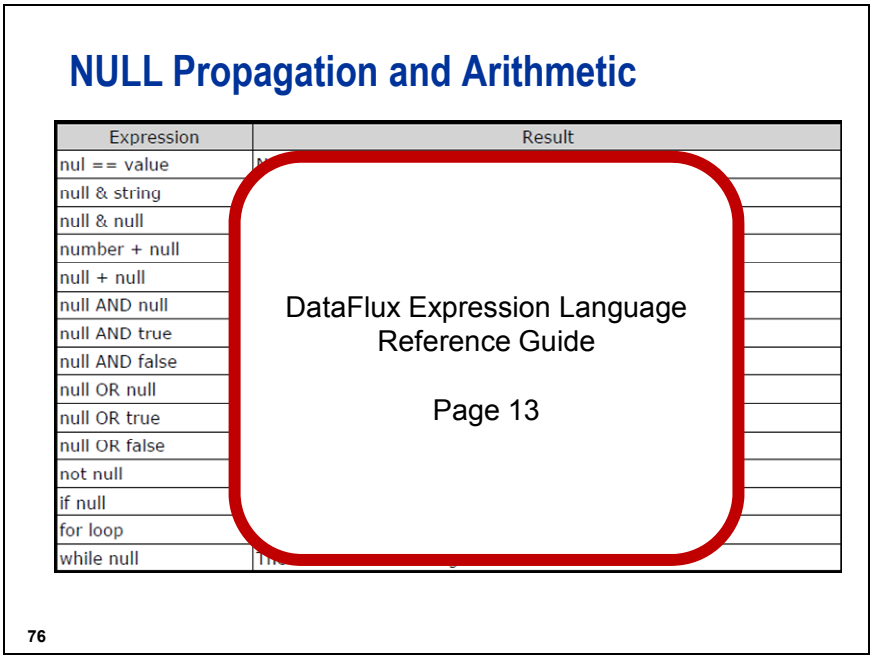
How Can You Find Empty Strings?

```
STRING name

name = "     "

IF (isblank(name) AND NOT isnull(name))
THEN
   print('Empty String')
ELSE
   IF isnull(name) THEN
      print('NULL')
   ELSE
      print('Not Empty String')
```

75

Using the isblank() and isnull() functions together with AND and NOT provides the solution to our problem. Look carefully at the IF/THEN/ELSE statements. There are two of them. They are nested.

The print() function arguments detail where the action happens. Take a look at the example and you can determine where the processing for each case takes place.



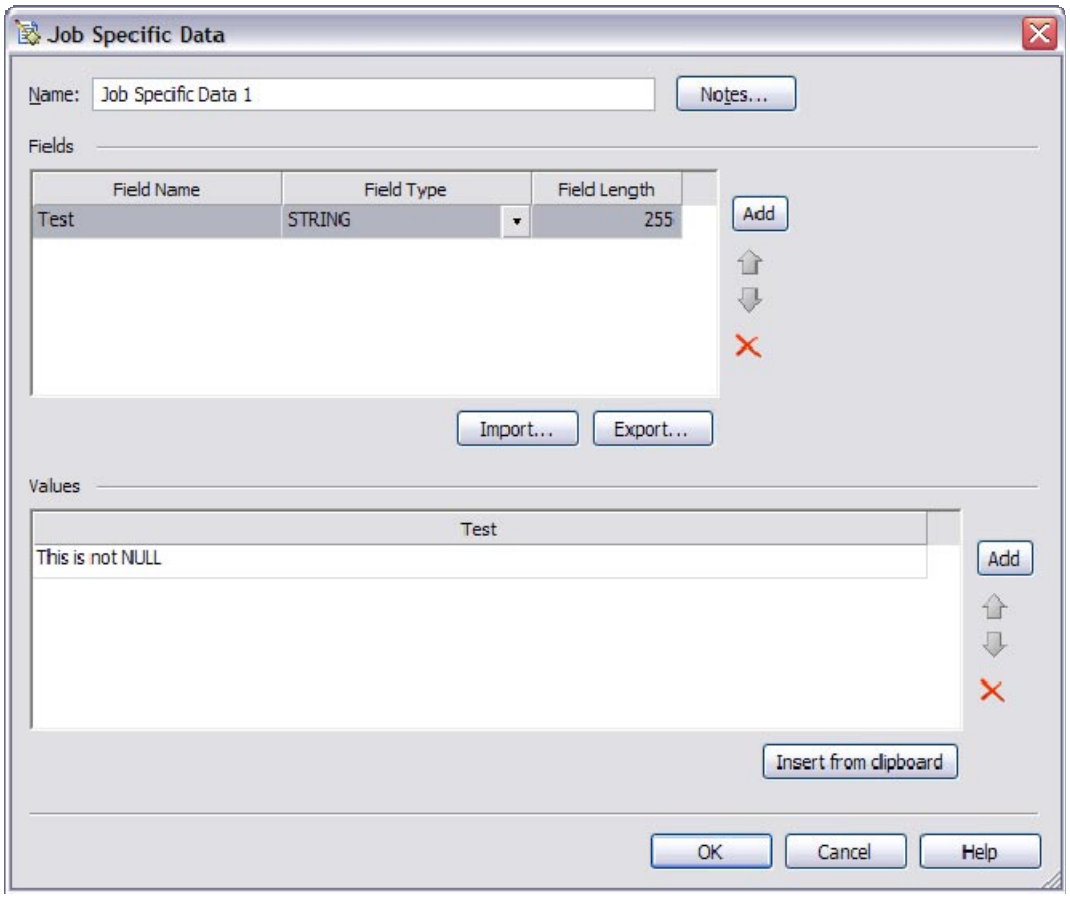
If you have NULL values in your data you need to know how arithmetic and logic expressions will behave. Usually, if any part of an expression is NULL then the entire expression is NULL. But, there are exceptions. Take a look at the DataFlux Expression Language Reference Guide for more information. You will find it on page 13.

Adding NULL Values to the Job Specific Data Node.

This is perhaps the only place you will find this documented!

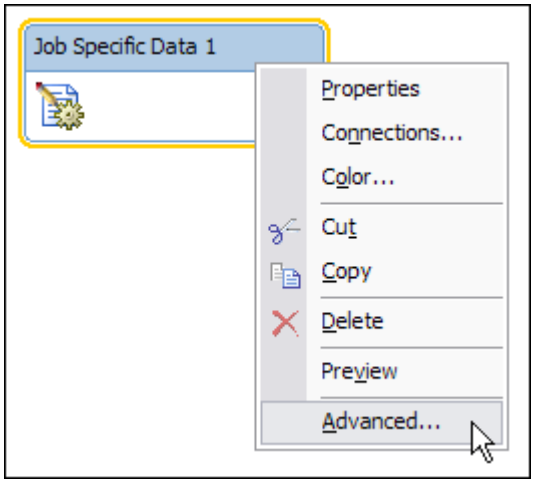
1. Invoke dfPower Studio by selecting **Start⇨All Programs⇨DataFlux dfPower Studio 8.1⇨dfPower Studio 8.1**.
2. Select  to invoke Architect.
3. Expand the Data Inputs hierarchy and drag and drop 

onto the page.
4. Double-click the Job Specific Data 1 node. Add a column named Test (leave the defaults: String of length 255). Add a value ("This is not a NULL") to the Test field.

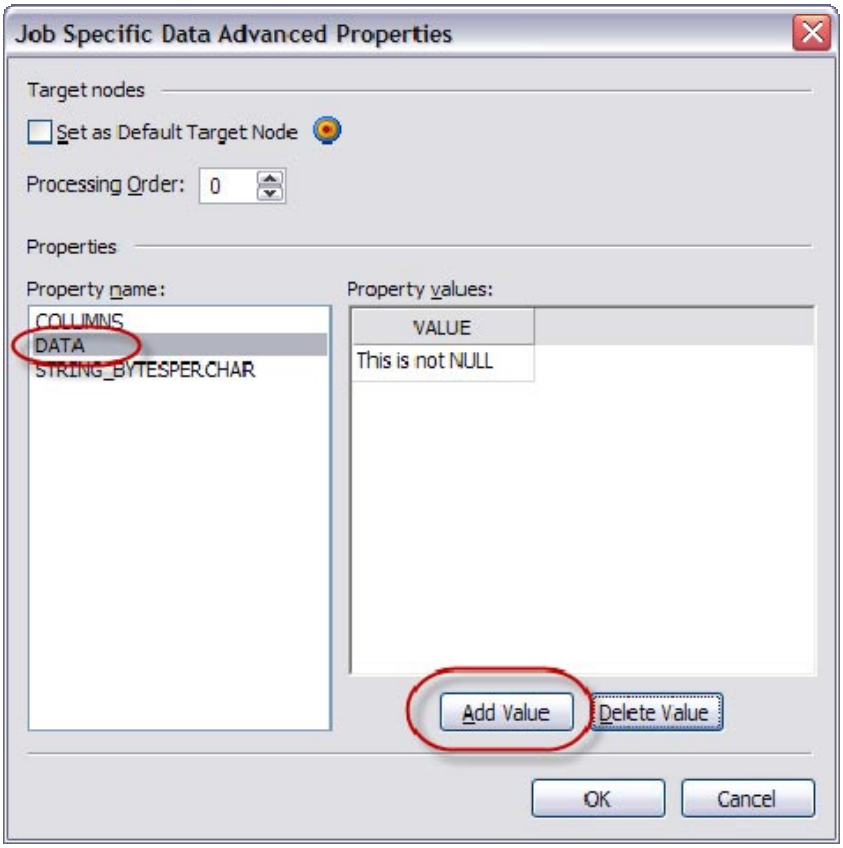


5. Select OK to close the dialog.

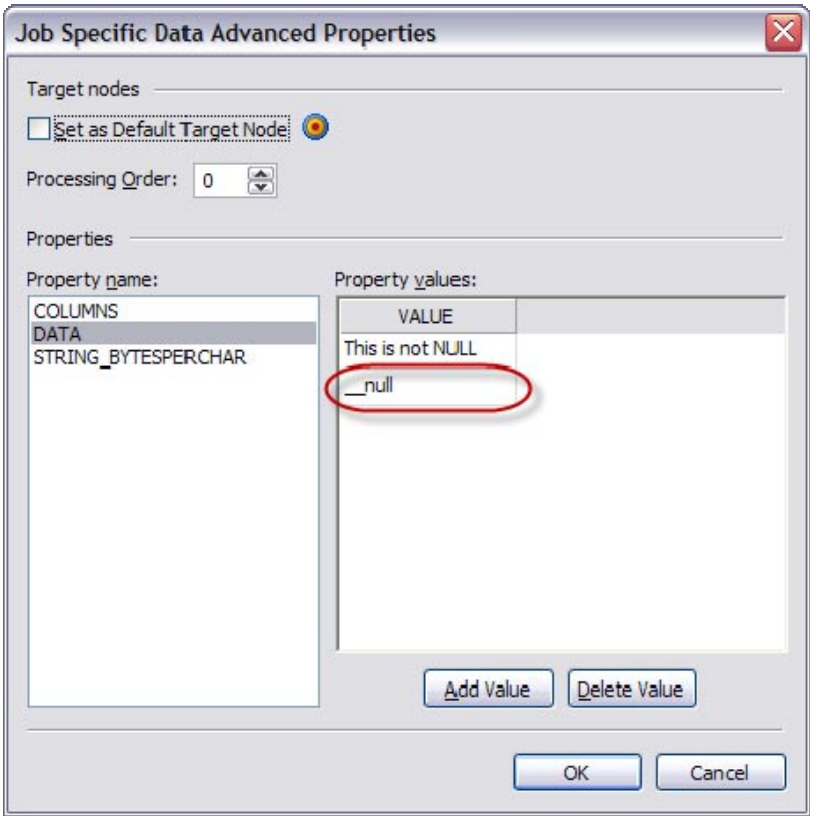
6. Right-click the Job Specific Data node. Select Advanced to open the Job Specific Data Advanced Properties dialog.

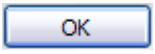


7. Select **DATA** under Property name then select Add Value.

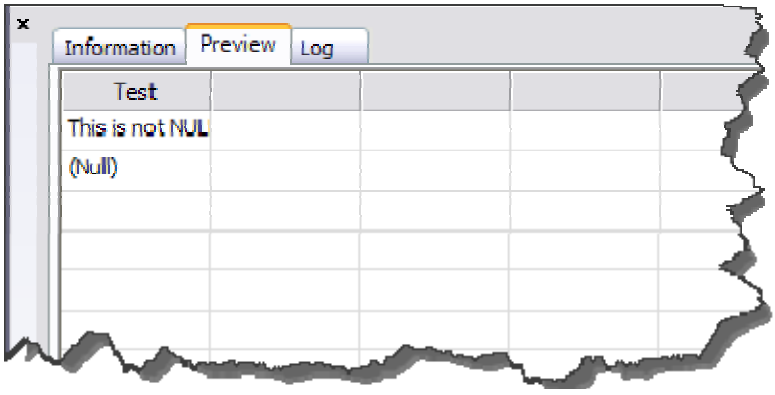


8. Enter **__null** in the Value field. Entering NULL there will create a string containing the "NULL."



9. Select  to close the Job Specific Data Advanced Properties dialog.

10. Look at the Preview tab on the dfPower Architect window. You should see (Null) listed under the Test field.



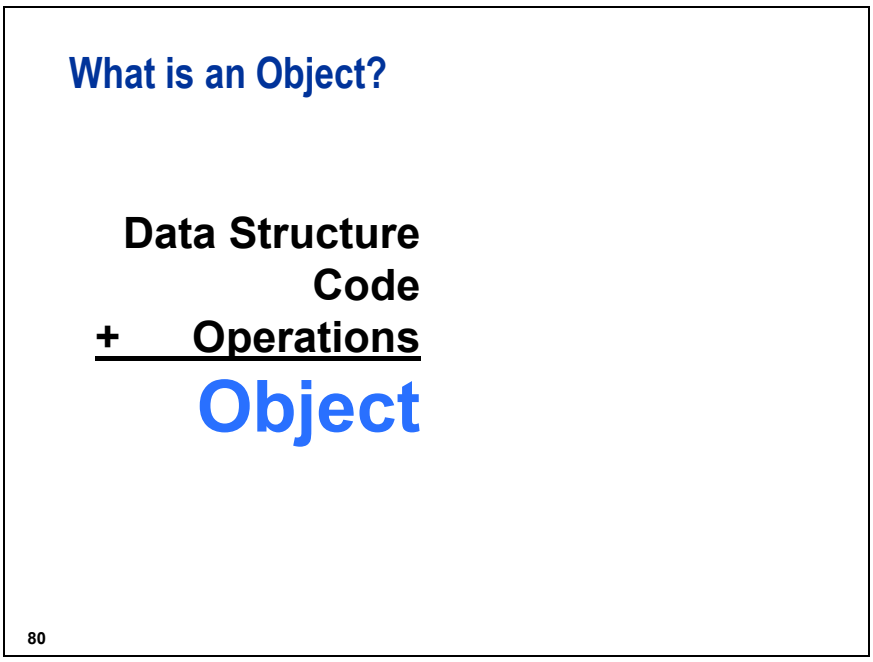
11. Close the dfPower Architect window.

12. It appears that if you add or change a value in the node the NULL value will disappear. Awesome! Just what we need – unnecessary confusion.

Exercises

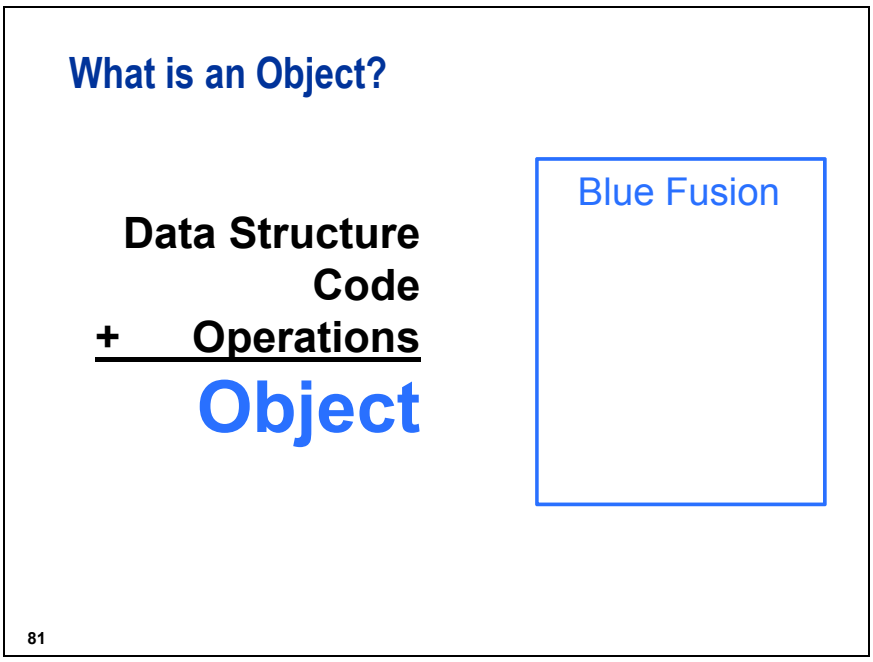
17. Create an expression language program that demonstrates the differences between empty, blank and NULL strings. Use a Job Specific Data node as input to the Expression node.

3.8 Objects



Objects in expression language are similar to other object oriented languages. And object combines data structures, code and operations into an easy to use programming construct. Although these objects are similar to other languages there is one difference: you can't create an object type in expression language.

There are 4 types of objects supported by the expression engine.

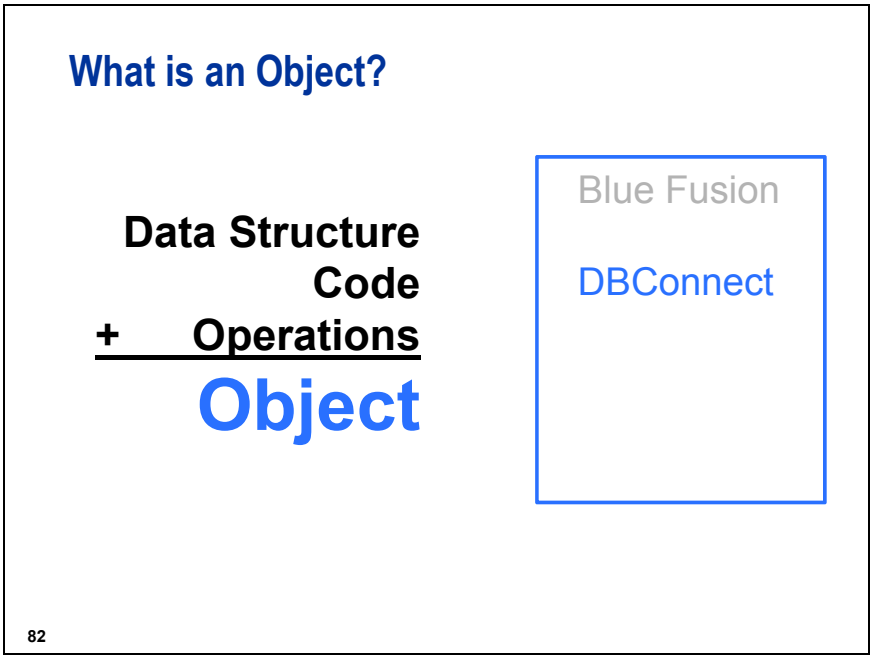


The Blue Fusion object allows you to access to internal DataFlux functionality.

You can use this object to

- generate match codes
- standardize input strings
- perform gender analysis
- analyze an input string and produce a pattern.

For more information Blue Fusion object, including coding examples, refer to the DataFlux Expression Language Reference Guide, page 87.

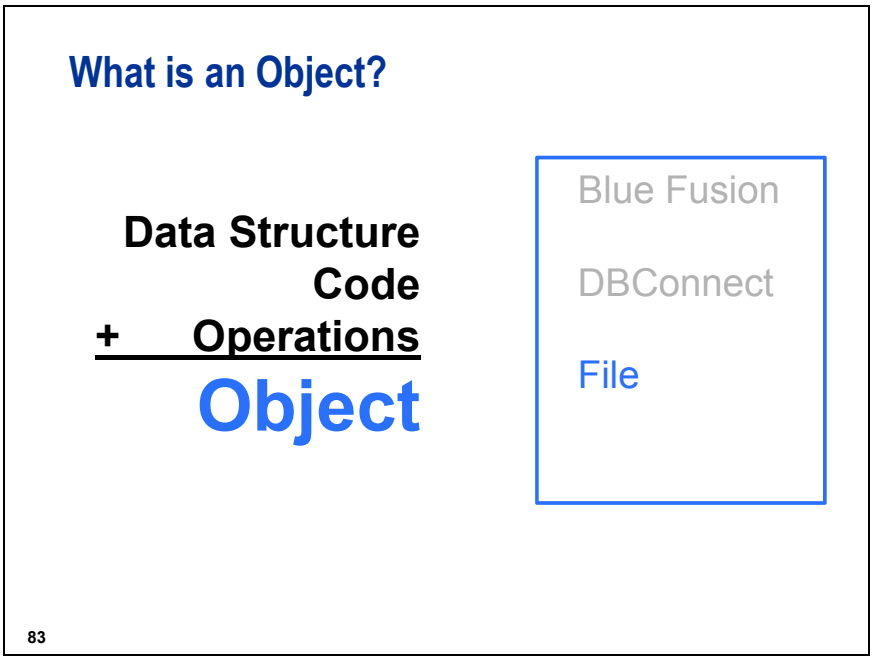


The DBConnect object allows you to interact with databases.

You can use this object to

- connect to a data source
- define a cursor
- get a list of all available data sources
- list tables
- execute a SQL query

For more information the DBConnect object, including coding examples, refer to the DataFlux Expression Language Reference Guide, page 80.

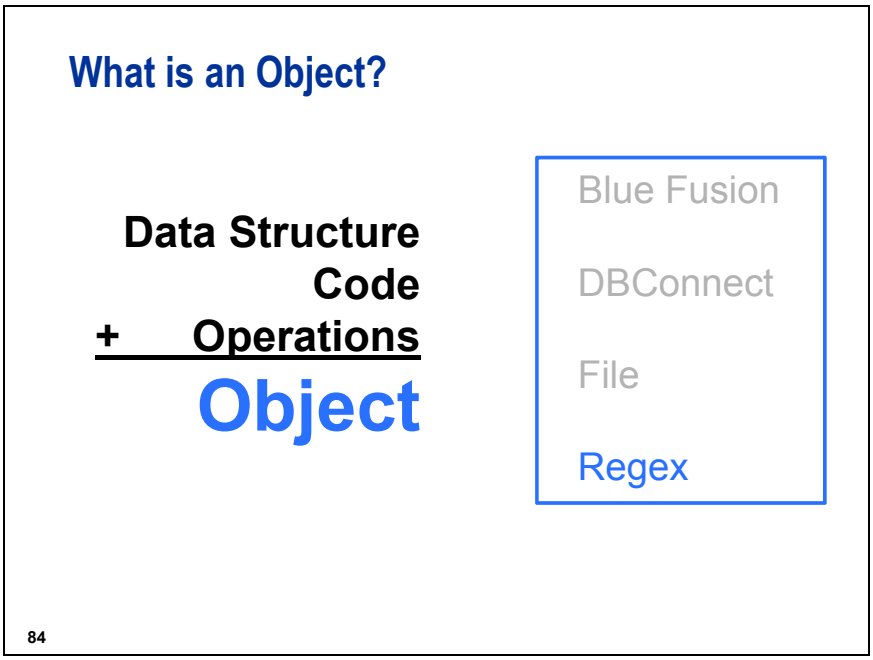


The File object allows you to interact with files.

You can use this object to

- read data from files
- write data to files
- create files
- move/copy files
- delete files

For more information on the File object, including coding examples, refer to the DataFlux Expression Language Reference Guide, page 65.



The Regex object allows you to interact with Perl Regular Expressions.

You can use this object to

- match strings
- replace strings
- find and count substrings

For more information on the Regular Expression object, including coding examples, refer to the DataFlux Expression Language Reference Guide, page 84.

3.9 Solutions

Solutions to Exercises

1. A – 4

B – 1

C – 2

D - 3

2. True

3. False

4. Create a symbol declaration statement for a symbol that can ...

a. Hold a 50 character string that can be seen in all expression blocks.

PUBLIC STRING(50) myString

b. Hold a number like 3.14 and can't be seen by child nodes.

PRIVATE REAL myReal

c. Hold a date

DATE myDate

d. Hold a 30 character string named FIRST NAME

STRING(30) `FIRST NAME`

5. Yes.

```
// Put code in the Pre-processing Expression tab
string(10) strNum
strNum = "1.11"

integer intNum
intNum = 1

real realNum
realNum = 2.22

real answer
answer = 0

answer = strNum + intNum + realNum
print(answer)
```

6. NULL

```
// Put code in the Pre-processing Expression tab
string(10) strNum
```

```
strNum = "Some String"

real realNum
realNum = 2.22

real answer
answer = 0

answer = strNum + realNum
print(answer)
```

7. The typeof() function returns the data type that an expression resolves to.

8. The problem is in the expression used in by the IF statement. = is an assignment operator. To fix this problem is should be == (two equal signs).

9. The last END statement has a space in it. Remove the space and problem is solved.

10. The problem is that there are no input records to read so the code on the Expression tab does not execute. Selecting the "Generate rows when no parent is specified" check box makes it appear to work but if you execute the job you will encounter an endless loop. Cut and paste the code into the Pre-Processing, or Post-processing tab will display the count in the Log tab.

11. Here is one way of doing this.

```
// put this code in the Pre or Post-processing Expression tab
boolean Continue_Looping
Continue_Looping = true

integer count
count = 1

while Continue_Looping
   begin
      print ("count --> " & count)
      count = count + 1
      if count <= 10 then
         Continue_Looping = true
      else
         Continue_Looping = false
   end
```

12. Here is one way of doing this.

```
// put this code in the Pre or Post-processing Expression tab
integer i

for i = 10 to 0 step -2
   begin
      print("count --> " & i)
   end
```

13. What do these functions do and what data type do they return? It is best if you write code and experiment with them.

isalpha - boolean isalpha(any). Returns true if the string is made up entirely of alphabetic characters.

isblank - boolean isblank(any). Returns true if expression is an empty string, NULL or contains only whitespace. The Functions dialog in dfPower Studio 8.1.1 does not mention that NULL strings are counted as blanks. The DataFlux Expression Language Reference Guide contains the correct information.

isnull - boolean isnull(any). Returns true if expression is null.

isnumber - boolean isnumber(any). Returns true if expression is a number.

left - string left(string, integer). Returns leftmost chars or a string.

len - integer len(string). Returns length of a string

locale - string locale(string). Set the locale. Returns the old locale. If no parameter is specified, the current locale is returned.

match_string - boolean match_string(string, string). Determine if the first string matches the second string, which may contain wildcards.

mid - string mid(string, integer, integer). Returns a sub-string of a string.

seteof - boolean seteof(boolean). Sets status to EOF, preventing further rows from being read from the step. If parameter is true, pushed rows will still be returned

upper - string upper(string). Returns the string in upper-case.

14. getvar()

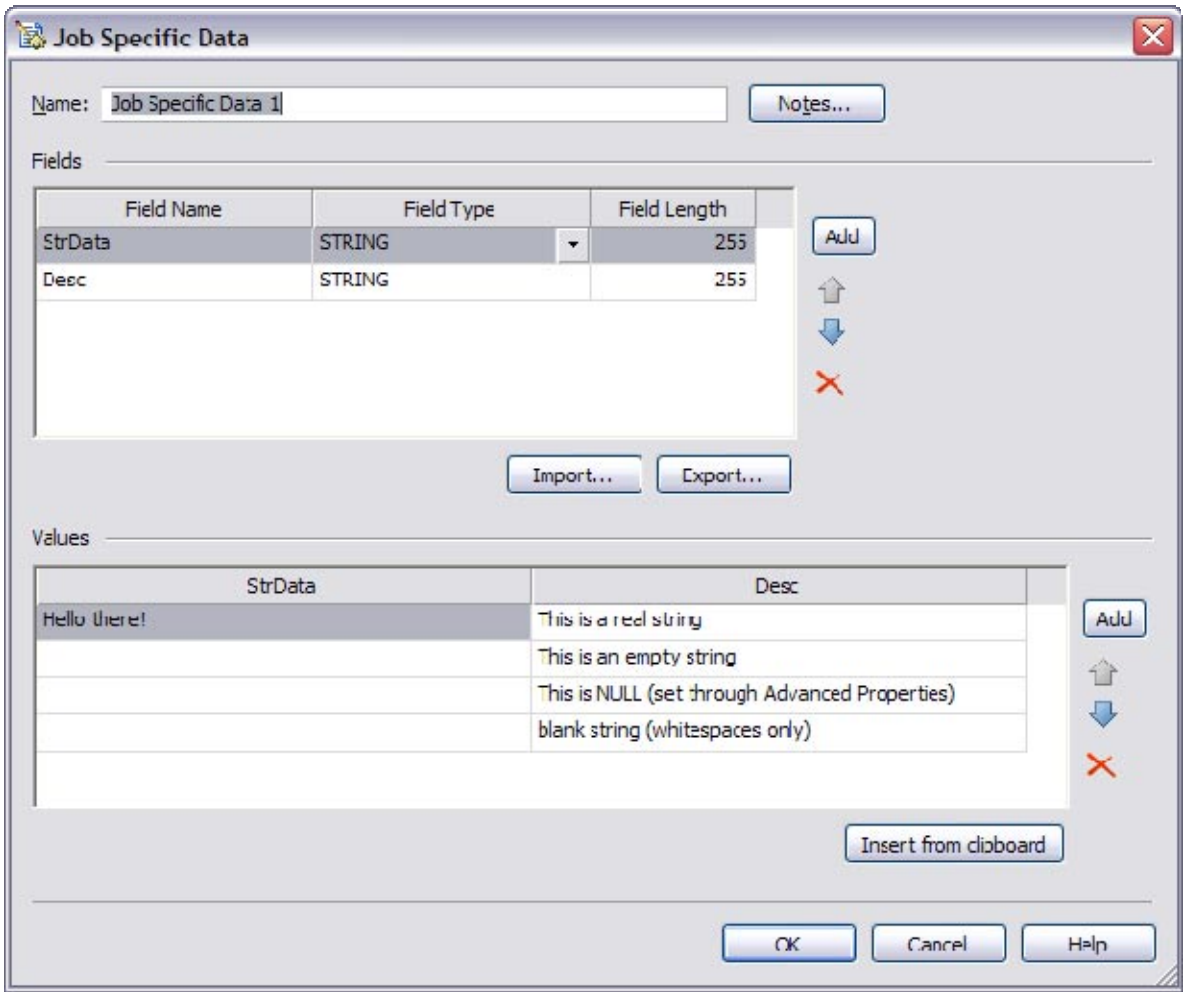
15. setvar()

16. Here is one possible solution. If you have problems with the Job Specific Data node take a look at the demo.

```
// Pre-processing Expression tab

string(30) array names

names.dim(11)

hidden integer row_cnt

row_cnt = 0


// Expression tab
row_cnt = row_cnt + 1
names.set(row_cnt, Name)


// Post-processing Expression
hidden integer index

for index = 1 to row_cnt step 1
   print("Index--> " & index & " Value--> " & Names.get(index))
```

17. The first dilemma is to put actual NULL values into the Job Specific Data node. That was covered in the demo. It wasn't required but I suggest adding a second column to the data. I named this column Desc. I use it to describe the contents of the StrData field that I created.

Here is the Job Specific Data node entries that I used. Keep in mind that you must use the Advance Properties tab to see the NULL values (__null).




```
// Expression tab
if isnull(StrData) then
   print("NULL value --> " & Desc)
else
   if isblank(StrData) then
      print("Empty or blank --> " & Desc)
   else
      print("Actual String Value --> " & Desc)
```