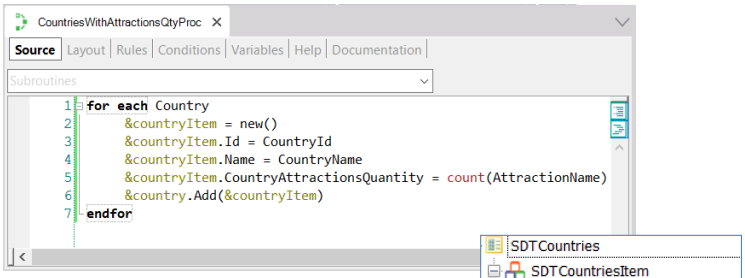
Logic for querying the database in GeneXus

For each command: an encompassing look

GeneXus™

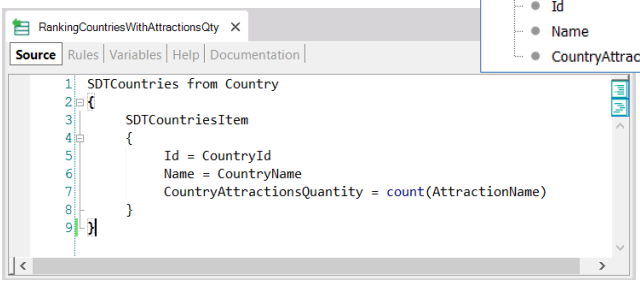
Here we will focus on reviewing the syntax of the For each command, with a view to extending its scope to other queries.

For each



Navigation groups

DP Group



Grids

In GeneXus, the For each command is the main way of accessing the database. This means that, in general, its logic will be valid for other forms of access, such as groups of data providers or grids with a base table in panels or web panels. To refer generically to any of these ways sometimes we use the expression "**navigation groups**."

Of course, there will be some differences; for example, the language of Data Providers is declarative and returns a structured output, so the body of a For each command is not programmed in the same way as a Data Provider group. However, the clauses that can be applied to it are almost the same... here we see the base transaction...

For each → *BaseTrn*
- **skip** *exp* **count** *exp*
- **order** *att…*
- **unique** *att…*
- **using** *DataSelector(parm…)*
- **where** *condition* when *condition*
- **blocking** *n*

Navigation groups

DP Group → *BaseTrn*
- **skip** *exp* **count** *exp*
- **order** *att…*
- **unique** *att…*
- **using** *DataSelector(parm…)*
- **where** *condition* when *condition*

Grids →
- **Base Trn** property
- **Order** property
- **Conditions** property
- **Unique** property
- **Data Selector** property

... but we also have all these others. The blocking one, as we will see, only applies to For Each commands (and to For each commands that update or delete).

Grids offer several of these clauses in properties, and what would correspond to the body of the For each is written in the corresponding Load event (also Refresh in the case of Panels).

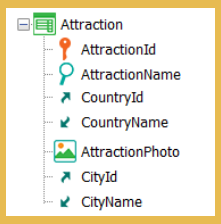
In addition, as we will review, although more restrictive, we have a query to the database when a Data Selector is executed in a Where clause with the In operator.

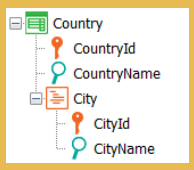
```
For each    BaseTrn1, … , BaseTrnn
        skip expression1 count expression2
        order  att1, att2, … , attn [when condition]
        order att1, att2, … , attn [when condition]
        order none [when condition]
        unique att1, att2, … , attn
        using DataSelector ( parm1, parm2, … , parmn )
        where  condition [when condition]
        where  condition [when condition]
        where  att  IN DataSelector ( parm1, parm2, … , parmn )
        blocking n
                main_code
        when duplicate
                when_duplicate_code
        when none
                when_none_code
endfor
```

So, let's review all the parts of the For each command and then go into some of them in more detail.

For each $BaseTrn_1, \ldots, BaseTrn_n$

```
For each Attraction
    print info //CountryName
endfor
```



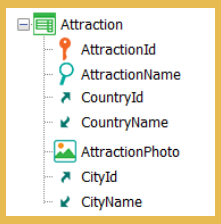


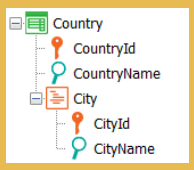
endfor

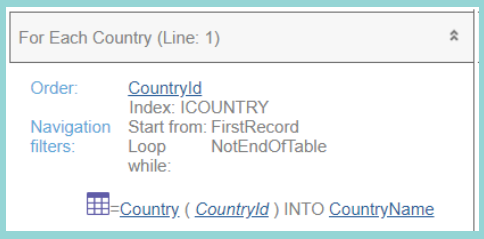
The base transaction is optional and is used to indicate which table should be the base table of the For each; that is, the table that will be accessed to return a set of records. In the previous courses we always used it, placing a single value there.
In this example, we are indicating that we want to run through all the attractions, and for each one we want to print the name of its country (for which it is also necessary to access the Country table; we see that this join is located on the server, that is, it will be resolved by the DBMS).

For each $BaseTrn_1, \ldots, BaseTrn_n$

```
For each Attraction
    print info //CountryName
endfor
```





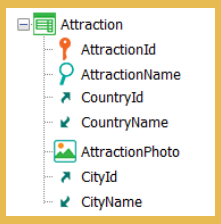


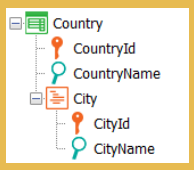
endfor

When no base transaction is specified, GeneXus must determine the table to navigate, according to the attributes indicated in the other places. In the example, as the only attribute inside the For each is CountryName, it will choose Country as the base table. Therefore, it will print all the country names of that table and here is a case where declaring a base transaction or not doing so changes the behavior.

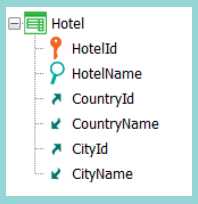
For each $BaseTrn_1, \ldots, BaseTrn_n$

```
For each Attraction
    print info //CountryName
endfor
```






```
For each Attraction, Hotel
    print info //AttractionName, HotelName
endfor
```



endfor

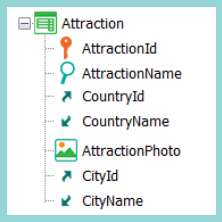
One case that hasn't been mentioned so far is when we specify more than one base transaction. There a Join or a Cartesian product will be made. In this example, where each attraction has a country and city and so does each hotel, a Join will be made.

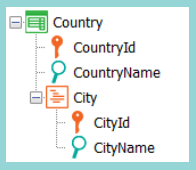
An attraction will be shown with a hotel of the same country and city, then the same attraction with another hotel of the same country and city, and so on until using all the hotels of the same country and city of the attraction, and moving on to the next attraction and doing the same thing again: it will be listed several times, but the hotel will change every time (from those of the same country and city). If they didn't have this relationship—for example, if the hotels didn't have a country and city—each attraction would be listed with each hotel in the table. Therefore, a Cartesian product will be made.

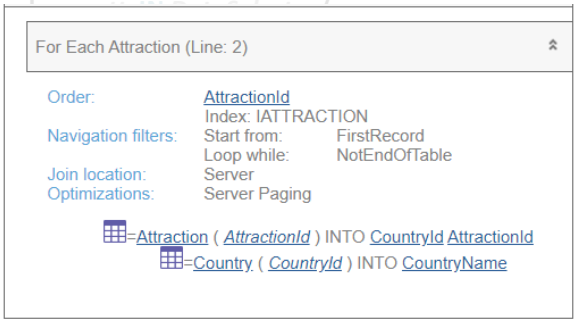
For each $BaseTrn_1, \dots, BaseTrn_n$

skip *expression*$_1$ **count** *expression*$_2$

```
For each Attraction
    skip 5 count 10

    print info //CountryName

endfor
```







Let's move on to the optional skip clause combined with count. It allows skipping the first n records (n being the result of evaluating this numeric expression) and from there keeping the following m records (m being the result of this other expression).

In this example, we skip the first 5 attractions of the query, in order to keep the next 10.

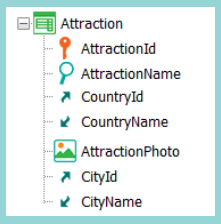
This is known as data paging. Paging consists basically of dividing the information resulting from a query into smaller blocks. Among other things, this reduces the amount of data sent from the database server to the program.

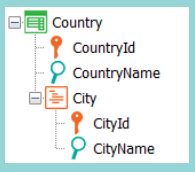
In the navigation list an optimization will be shown, where we can see that this paging will be done on the server; that is, the DBMS itself will do it. We don't need to worry about it.

For each $BaseTrn_1$, ... , $BaseTrn_n$

skip *expression*$_1$ **count** *expression*$_2$
order *att*$_1$, *att*$_2$, ... , *att*$_n$ [**when** *condition*]
order *att*$_1$, *att*$_2$, ... , *att*$_n$ [**when** *condition*]
order none [**when** *condition*]
unique *att*$_1$, *att*$_2$, ... , *att*$_n$
using *DataSelector* (*parm*$_1$, *parm*$_2$, ... , *parm*$_n$)
where *condition* [**when** *condition*]
where *condition* [**when** *condition*]
where *att* **IN** *DataSelector* (*parm*$_1$, *parm*$_2$, ... , *parm*$_n$)

```
For each Attraction
    order CountryId when not &country.IsEmpty()
    order AttractionName
    where CountryId >= &country when not &country.IsEmpty()

            print info //AttractionName, CountryName

endfor
```





We could then specify a list of conditional order clauses with a maximum of one unconditional clause at the end, to sort the information to be queried and returned. Conditions also allow us to play with conditional where clauses, in order to sort by the same filter criteria, and to specify more optimized queries.

In this example, if the condition of the first order clause is met—that is, the &country variable is not empty—it is sorted by CountryId and the following order clause is not considered at all. Since in this case the where condition is the same, the where will be applied and the country filter will then be optimized.

On the other hand, if the condition of the first order clause is not met, the second order clause is looked into, and since it is unconditional it will apply. As the where will not apply in this case, because the &country variable is empty, all attractions will be displayed sorted by attraction name (and most likely not ordered by country).

The order none clause, which we hadn't seen before, is written when we want to leave the order to be applied undefined, which means that it will depend on the platform and may even vary from one execution to the next.

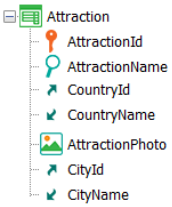
Anyway, the order clauses that we write do not determine the exact execution plan that the DBMS will choose, because other considerations are also taken into account, precisely to optimize the database access. We will explore this topic in more detail in another video.

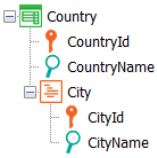
We also know that the order clause plays a fundamental role when implementing

a control break between nested For each commands. In that case, it is not only used to order the information but also, and more importantly, to establish the break criteria. There the order clause cannot be conditional.

For each *BaseTrn$_1$, … , BaseTrn$_n$*

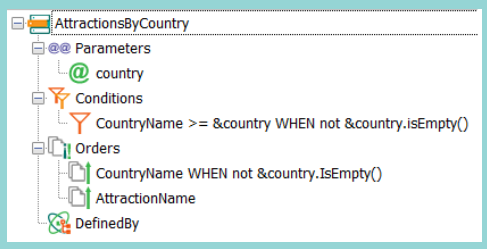
skip *expression$_1$* **count** *expression$_2$*
order *att$_1$, att$_2$, … , att$_n$* **[when** *condition***]**
order *att$_1$, att$_2$, … , att$_n$* **[when** *condition***]**
order none [when *condition***]**
unique *att$_1$, att$_2$, … , att$_n$*
using *DataSelector* **(** *parm$_1$, parm$_2$, … , parm$_n$* **)**






```
For each Attraction
    order CountryName when not &country.IsEmpty()
    order AttractionName
    where CountryName >= &country when not &country.IsEmpty()
            print info //AttractionName, CountryName
endfor
```

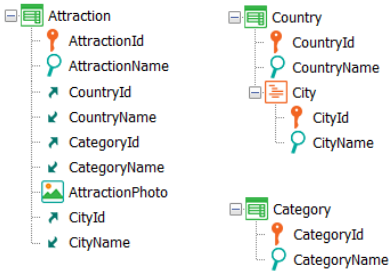
```
For each Attraction
    using AttractionsByCountry(&country)
            print info //AttractionName, CountryName
endfor
```



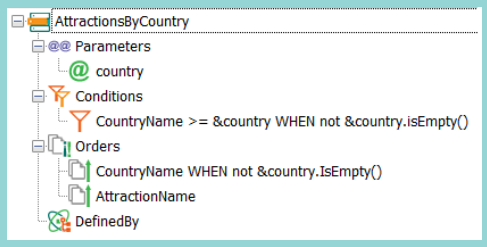
The using clause allows us to incorporate more orders and filters but centralized in a Data Selector object, which makes it possible to send parameters to it. In this way, we don't have to repeat those same sort and filter criteria explicitly in each query. As far as the For each is concerned, it's as if they have been written explicitly. It makes no difference. In fact, in the navigation list it will not be possible to differentiate one form from the other. It will look exactly the same and there will be no indication of the Data Selector.

For each *BaseTrn$_1$, … , BaseTrn$_n$*

skip *expression$_1$* **count** *expression$_2$*
order *att$_1$, att$_2$, … , att$_n$* **[when** *condition***]**
order *att$_1$, att$_2$, … , att$_n$* **[when** *condition***]**
order none [when *condition***]**
unique *att$_1$, att$_2$, … , att$_n$*
using *DataSelector* **(** *parm$_1$, parm$_2$, … , parm$_n$* **)**



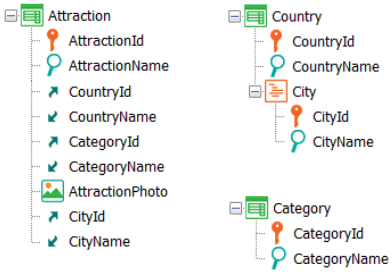

```
For each Attraction
    order CountryName when not &country.IsEmpty()
    order AttractionName
    where CountryName >= &country when not &country.IsEmpty()
    where CategoryName <> "Monument"
            print info //AttractionName, CountryName
endfor

For each Attraction
    using AttractionsByCountry(&country)
    where CategoryName <> "Monument"
            print info //AttractionName, CountryName
endfor
```



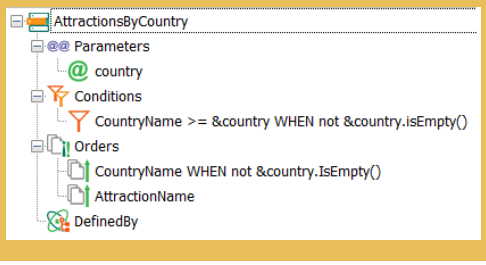
This means that order, where, and using clauses can coexist without any problem. For the generated program, there will be no difference between this For each and this other one.

For each *BaseTrn$_1$, … , BaseTrn$_n$*

skip *expression$_1$* **count** *expression$_2$*
order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]
order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]
order none [**when** *condition*]
unique *att$_1$, att$_2$, … , att$_n$*
using *DataSelector* (*parm$_1$, parm$_2$, … , parm$_n$*)
where *condition* [**when** *condition*]
where *condition* [**when** *condition*]
where *att* **IN** *DataSelector* (*parm$_1$, parm$_2$, … , parm$_n$*)
blocking *n*
main_code
when duplicate
when none
when_none_code
endfor




```
For each Category
    where CategoryId IN AttractionsByCountry(&country)
            print info //CategoryName
endfor
```



The difference comes up when the Data Selector is used in another way, this time in a special way: when it is used as an executable query.

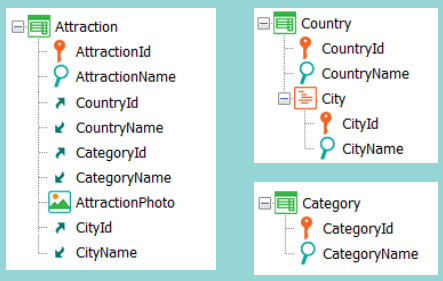
This happens when we want to filter indicating that we only want to keep the records of the extended table of the For each for which the value of a certain attribute is among those returned by this independent query.

In the example, when we want to run through the categories table keeping only those that are among the attraction categories of the independent query of the Data Selector.
The DataSelector query, because of the attributes involved, is clearly of the attractions of countries with names after the filter (if not empty, if empty, all). In this case, we should consider the DataSelector as if it were another For each. This is why the attributes inside the Data Selector for this case will not have any relevance for the For each where it is being used. Thus, the base table of the For each will be Category, and that of the query of the Attraction Data Selector.

For each *BaseTrn$_1$, … , BaseTrn$_n$*

skip *expression$_1$* **count** *expression$_2$*
order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]
order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]
order none [**when** *condition*]
unique *att$_1$, att$_2$, … , att$_n$*
using *DataSelector* (*parm$_1$, parm$_2$, … , parm$_n$*)
where *condition* [**when** *condition*]

```
For each Attraction
    unique CategoryId
        print info //CategoryName
endfor

For each Attraction
    unique CategoryId
        &qty = count( AttractionName )
        print info //CategoryName &qty
endfor
```



We also saw the unique clause, so that if the indicated attributes are repeated for a set of records in the For each, only one is taken for processing in the body of the For each; that is, in the main code.

Here we are running through the table of attractions, grouping them by category ID and only keeping one record from the group, printing its category name.

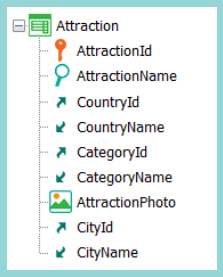
We also saw that this clause was extremely useful when in the code of the For each we wanted to do an aggregation on that set of repeated records for those unique attributes. Something that otherwise we could not achieve (having the For each and the aggregation formula working on the same table). In this example, the For each runs through Attraction, and so does the Count formula. In addition, since we have specified the unique clause by CategoryId, the count will be made for the set of records with the same CategoryId in which it is positioned each time.

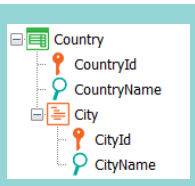
For each *BaseTrn$_1$, … , BaseTrn$_n$*

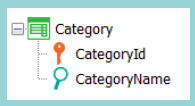
skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
For each Category
   ...
      print info //CategoryName
      For each Attraction
         print info2 // AttractionName, CountryName, CityName
      endfor
endfor
```







main_code

when duplicate

when_duplicate_code

when none

when_none_code

endfor

In the main code block we program what we want to do with each record of the base table that has passed the filters (of course, if attributes of the extended table are used there, these records are automatically accessed and used).

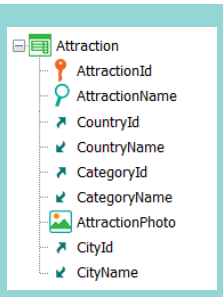
What can be programmed here includes writing another For each command, nested...

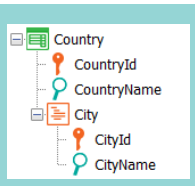
For each *BaseTrn$_1$, … , BaseTrn$_n$*

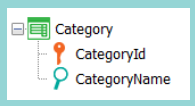
skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
For each Category
   …
      print info //CategoryName
      Do 'Something'
      print info3

endfor
```







main_code

when duplicate

when_duplicate_code

when none

when_none_code

endfor

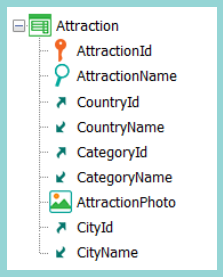
...an invocation to a subroutine or...

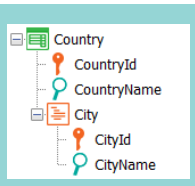
For each *BaseTrn$_1$, … , BaseTrn$_n$*

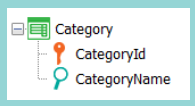
skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
For each Category
   ...
      print info //CategoryName
      MyProcedure( CategoryId )
      print info3

endfor
```







main_code

when duplicate

when_duplicate_code

when none

when_none_code

endfor

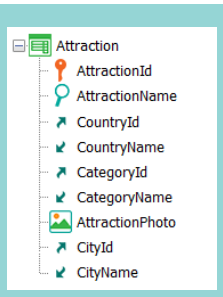
...to a procedure, which, when executed, will return to whatever follows the invocation.

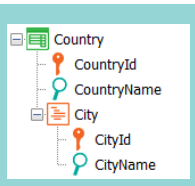
For each *BaseTrn$_1$, … , BaseTrn$_n$*

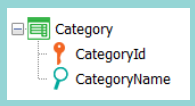
skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
For each Attraction
    where CountryName = "France"
        print info //AttractionName, CategoryName
    when none
        print infoNone      //"No attractions from France"
endfor
```







main_code

when duplicate

when_duplicate_code

when none

when_none_code

endfor

If no record passes the filters, then the code specified under the when none clause will be executed—if it was specified, of course.

In this example, it will be when there is no attraction from a country of that name.

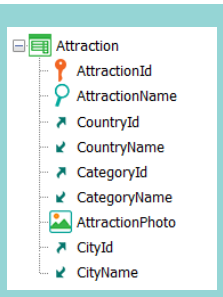
Since it is assumed that nothing was done with the data in the extended table of the For each when this clause was executed, here we are not positioned in any record, so....

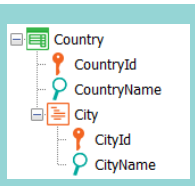
For each *BaseTrn$_1$, … , BaseTrn$_n$*

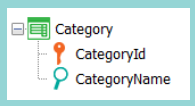
skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
parm(in:CategoryName);

For each Attraction
    where CountryName = "France"
        print info //AttractionName, CategoryName
    when none
        print infoNone      //CategoryName
endfor
```







main_code

when duplicate

when_duplicate_code

when none

when_none_code

endfor

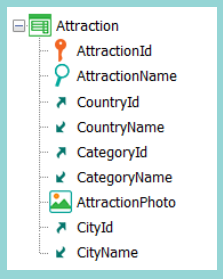
...if attributes are named there, where will they be taken from? It only makes sense to name attributes if they have already been instantiated (for example, in the parm rule of the object where this For each is located....

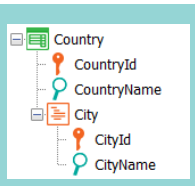
For each *BaseTrn$_1$, … , BaseTrn$_n$*

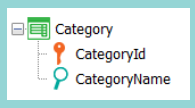
skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
For each Category
    For each Attraction
        where CountryName = "France"
            print info //AttractionName, CategoryName
        when none
            print infoNone      //CategoryName
    endfor
endfor
```







main_code

when duplicate

when_duplicate_code

when none

when_none_code

endfor

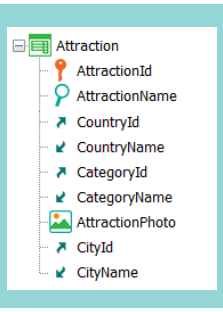
... or in another For each in which this one is nested).

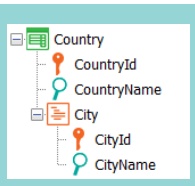
For each *BaseTrn$_1$, … , BaseTrn$_n$*

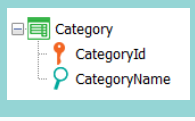
skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
For each Attraction
    where CountryName = “France”
        print info //AttractionName, CategoryName
    when none
        For each Country.City
            print info2 //CountryName, CityName
        endfor
endfor
```








```
For each Attraction
    where CountryName = “France”
        print info //AttractionName, CategoryName
endfor
For each Country.City
    print info2 //CountryName, CityName
endfor
```

Of course, here we can write another query (another For each), use an inline formula, etc., but it is as if they were written after the For each.

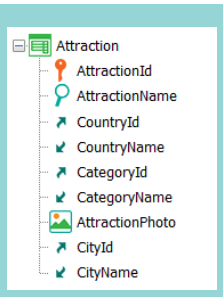
For each *BaseTrn$_1$, … , BaseTrn$_n$*

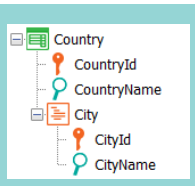
Procedure Object

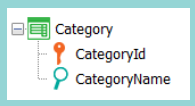
skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
For each Attraction
   where CountryName = "France"

      AttractionName = "France " + AttractionName
      CategoryName = "France Attractions"

endfor
```







main_code

when duplicate

when_duplicate_code

when none

when_none_code

endfor

If the For each is inside a procedure, and only in that case, in the main code it will also be possible to assign a value to one or several attributes, in order to update the record of the base table and the corresponding record(s) of the extended table.

Here we update the AttractionName attribute of the record of the base table in which we are in each iteration, and the CategoryName attribute of the associated record of the Category table, which is part of the extended one.

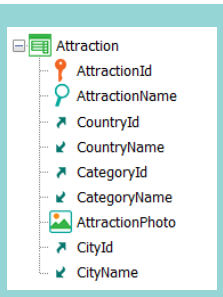
For each *BaseTrn$_1$, … , BaseTrn$_n$*

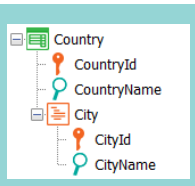
Procedure Object

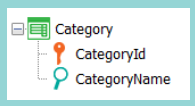
skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
For each Attraction
   where CountryName = "France"

         Delete

endfor
```







main_code

when duplicate

when_duplicate_code

when none

when_none_code

endfor

We can also write a Delete command to delete the record on which we are positioned (and it only deletes the record from the base table, without performing referential integrity checks).

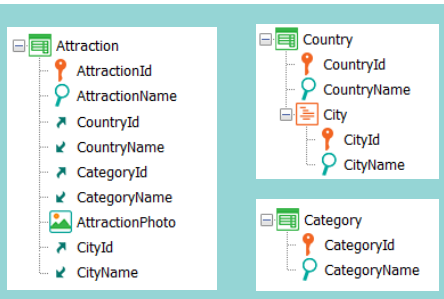
For each *BaseTrn$_1$, … , BaseTrn$_n$*

Procedure Object

skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
For each Attraction
    where CountryName = “France”
        new
            CategoryName = “Famous Landmark”
        endnew
endfor
```



main_code

when duplicate

when_duplicate_code

when none

when_none_code

endfor

Of course, we can also write a new command to insert a record into a table, but this goes beyond the behavior of the For each, which controls both updates and deletions.

For each *BaseTrn$_1$, … , BaseTrn$_n$*

skip *expression$_1$* **count** *expression$_2$*

order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]

```
For each Attraction
    where CountryName = "France"
     blocking 10
        AttractionName = "France " + AttractionName
        CategoryName = "France Attractions"
endfor
```

main_code

when duplicate

when_duplicate_code

when none

when_none_code

endfor

Procedure Object

Syntax

```
New
   [Defined by attributeList]
   [Blocking NumericExpression]
             BodyCode
[When duplicate
   { AnotherCode |
       For each
         {att = exp}
         …
       Endfor
     | AnotherCode } ]
 EndNew
```

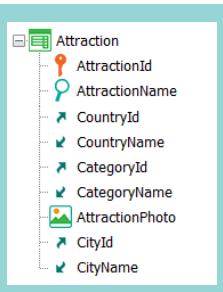
For either of these two cases we have the blocking clause. It allows the operations to be performed in a buffer and when the n records indicated in the clause are processed (in this example 10), only then are the operations performed on the database. That is, the database is accessed only once to process the n records that are being updated or deleted, thus improving performance.

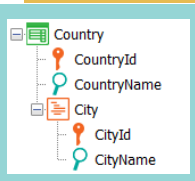
As expected, this same clause is also available in the New command.

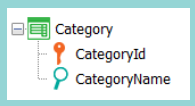
For each $BaseTrn_1, \ldots, BaseTrn_n$

Procedure Object

```
For each Attraction
    where CountryName = "France"
        AttractionName = "France " + AttractionName
        CategoryName = "France Attractions"
    when duplicate
        For each
            AttractionName = "France " + AttractionName
        endfor
endfor
```







AttractionId	AttractionName	CountryId	CityId	CategoryId
1	Louvre Museum	2	1	1
2	The Great Wall	3	1	2
3	Eiffel Tower	2	1	2
4	Forbidden city	3	1	2
5	Christ the Redeemer	1	2	2

CategoryId	CategoryName
1	Museum
2	Monument
3	Tourist site
4	France Attractions
5	Famous Landmark

endfor

Finally, the when duplicate clause is used when you want to update attributes in the body of the For each that are, in particular, candidate keys, so their values must not be repeated.

If in the main code the value of one of these attributes is being modified (for the record in which the program is positioned in that iteration), assigning it a value that already exists for another record in the table, then the update will not be allowed (remember that the unique index is used to perform this control).

Let's imagine for this case that in the Category table there already exists a record with CategoryName “France Attractions,” and that there is a unique index by CategoryName. So, for attraction 1, which is from France, when trying to update the value of its CategoryName, which is Museum, to “France Attractions,” since a uniqueness control will be made, the check will fail due to a duplicated key. There is already a record with the value 4.

If the developer programmed a when duplicate clause, that is where he or she indicates what should be done in that case. If the clause is not written, nothing will be done with that record that was intended to be updated and it moves on to the next iteration of the For each, where in this case the same will happen when trying to modify the category name of the Eiffel Tower, which is 2.

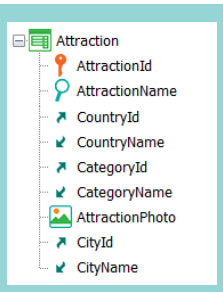
If the when duplicate clause is specified, then although we may think that we are still positioned in the record that caused the problem, we are no longer positioned to update, so if we want to update the attribute or another one with another value, we have to write a For each to indicate it.

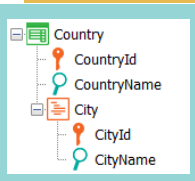
In this case, we want the first update (which will never fail) to be performed, and... for the first attraction of France, it fails due to a duplicated one...

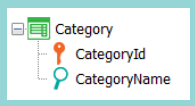
For each $BaseTrn_1, \dots, BaseTrn_n$

Procedure Object

```
For each Attraction
    where CountryName = "France"
        AttractionName = "France " + AttractionName
        CategoryName = "France Attractions"
    when duplicate
        For each
            AttractionName = "France " + AttractionName
        endfor
endfor
```







AttractionId	AttractionName	CountryId	CityId	CategoryId
1	France Louvre Museum	2	1	1
2	The Great Wall	3	1	2
3	Eiffel Tower	2	1	2
4	Forbidden City	3	1	2
5	Christ the Redeemer	1	2	2

CategoryId	CategoryName
1	Museum
2	Monument
3	Tourist site
4	France Attractions
5	Famous Landmark

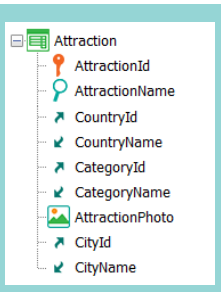
endfor

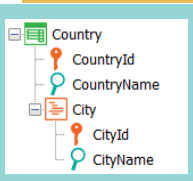
…and executes this section. Next, for the second and last attraction in France: it tries to update and also fails...

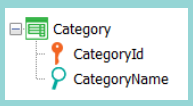
For each $BaseTrn_1, \dots, BaseTrn_n$

Procedure Object

```
For each Attraction
    where CountryName = "France"
        AttractionName = "France " + AttractionName
        CategoryName = "France Attractions"
    when duplicate
        For each
            AttractionName = "France " + AttractionName
        endfor
endfor
```







AttractionId	AttractionName	CountryId	CityId	CategoryId
1	France Louvre Museum	2	1	1
2	The Great Wall	3	1	2
3	France Eiffel Tower	2	1	2
4	Forbidden City	3	1	2
5	Christ the Redeemer	1	2	2

CategoryId	CategoryName
1	Museum
2	Monument
3	Tourist Site
4	France Attractions
5	Famous Landmark

endfor

…then it runs this section, and the table will look like this.

For each *BaseTrn$_1$, … , BaseTrn$_n$*

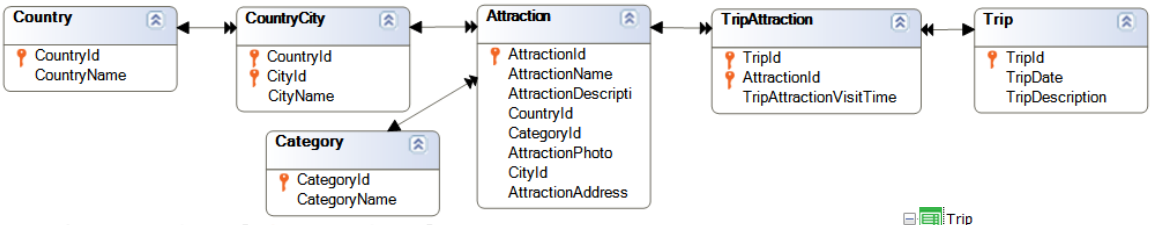
skip *expression$_1$* **count** *expression$_2$*
order *att$_1$*, *att$_2$*, … , *att$_n$* [**when** *condition*]
order *att$_1$*, *att$_2$*, … , *att$_n$* [**when** *condition*]
order none [**when** *condition*]
unique *att$_1$*, *att$_2$*, … , *att$_n$*
using *DataSelector* (*parm$_1$*, *parm$_2$*, … , *parm$_n$*)
where *condition* [**when** *condition*]
where *condition* [**when** *condition*]
where *att* **IN** *DataSelector* (*parm$_1$*, *parm$_2$*, … , *parm$_n$*)
blocking *n*
main_code
when duplicate
when_duplicate_code
when none
when_none_code

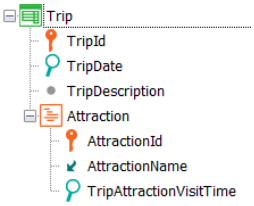
endfor

In short, all the clauses prior to the main code or body of the For each are used to filter the records on which the main code will be executed, to sort them, group them based on repeated values, and process them once, or to indicate that they will be accessed as a block when you want to update and/or delete them.

Then, for each one of the records that pass these filters or the grouping, and according to the determined order, the main code will be executed.

From the attributes used in all these clauses as well as within the main code, it is often necessary to access records from other tables of the extended table, but not all of them.
For performance reasons, only those will be retrieved and this has some consequences.






```
For each Trip.Attraction                    parm( in: CategoryName);
   order AttractionName
   where CountryName = “France”
   where TripDate >= &today

      print info //AttractionName, TripAttracitionVisitTime

endfor
```

In this example, the TripAttraction table will be run. It corresponds to the second level of the Trip transaction, but Attraction will be accessed to sort by AttractionName and because its value has to be printed in the output. Also, because from there CountryCity will be accessed in order to access Country to be able to filter by CountryName.
In addition, Trip will be accessed in order to filter by TripDate.
But it is not necessary at all to access Category. Therefore, if in the parm rule we receive in the CategoryName attribute, contrary to what we might think, that filter will not be applied. Why? Because inside the For each the Category table is not accessed at all. So, after determining all the navigations of the object where this For each is located, when the implicit filter that comes from the parm rule is going to be added, inspecting each of the queries, no relationship is found for this For each and it is not added.

Procedure Object

For each *BaseTrn$_1$, … , BaseTrn$_n$*

skip *expression$_1$* **count** *expression$_2$*
order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]
order *att$_1$, att$_2$, … , att$_n$* [**when** *condition*]
order none [**when** *condition*]
unique *att$_1$, att$_2$, … , att$_n$*
using *DataSelector* (*parm$_1$, parm$_2$, … , parm$_n$*)
where *condition* [**when** *condition*]
where *condition* [**when** *condition*]
where *att* **IN** *DataSelector* (*parm$_1$, parm$_2$, … , parm$_n$*)
blocking *n*
main_code
when duplicate
when_duplicate_code
when none
when_none_code

endfor

```
For each Attraction
   where CountryName = "France"
      AttractionName = "France " + AttractionName
      CategoryName = "France Attractions"
   when duplicate
      For each
         AttractionName = "France " + AttractionName
      endfor
endfor
```

For the case in which we are in a procedure and in the main code we want to update attributes that are candidate keys, the following happens:
Let's suppose that we are processing the record n that passes the filters; that is, we are in iteration n of the For each, where the code for the previous n-1 records was already executed, and for that record or one of its related records by extended table we are trying to update one of its attributes that is a candidate key by giving it a repeated value. Then if there is no when duplicate clause, the update is not performed and it moves on to the next record of the iteration, n+1. But if the when duplicate clause is specified, it is executed. Then the next record of the For each, n+1, is processed.

For each *BaseTrn$_1$, … , BaseTrn$_n$*

skip *expression$_1$* **count** *expression$_2$*
order *att$_1$*, *att$_2$*, … , *att$_n$* [**when** *condition*]
order *att$_1$*, *att$_2$*, … , *att$_n$* [**when** *condition*]
order none [**when** *condition*]
unique *att$_1$*, *att$_2$*, … , *att$_n$*
using *DataSelector* (*parm$_1$*, *parm$_2$*, … , *parm$_n$*)
where *condition* [**when** *condition*]
where *condition* [**when** *condition*]
where *att* **IN** *DataSelector* (*parm$_1$*, *parm$_2$*, … , *parm$_n$*)
blocking *n*
main_code
when duplicate
when_duplicate_code
when none
when_none_code

endfor

The code in the when none clause will only be executed when there was no iteration of the For each since no record passed the filters (or the table was empty). That is, if this code is executed, it is because none of these was executed for any record.

For each *BaseTrn$_1$, … , BaseTrn$_n$*

skip *expression$_1$* **count** *expression$_2$*
order *att$_1$*, *att$_2$*, … , *att$_n$* [**when** *condition*]
order *att$_1$*, *att$_2$*, … , *att$_n$* [**when** *condition*]
order none [**when** *condition*]
unique *att$_1$*, *att$_2$*, … , *att$_n$*
using *DataSelector* (*parm$_1$*, *parm$_2$*, … , *parm$_n$*)
where *condition* [**when** *condition*]
where *condition* [**when** *condition*]
where *att* **IN** *DataSelector* (*parm$_1$*, *parm$_2$*, … , *parm$_n$*)
blocking *n*
main_code
when duplicate
when_duplicate_code
when none
when_none_code

endfor

As we have just seen, the attributes that appear in the code of when duplicate or when none, as well as those that appear inside the Data Selector when it is executed as an independent query, are not considered at all when determining the navigation of the For each.

This is especially important when GeneXus must determine the base table of the For each, because no base transaction was placed.

This is the end of this summary of essential concepts.

training.genexus.com
wiki.genexus.com
training.genexus.com/certifications