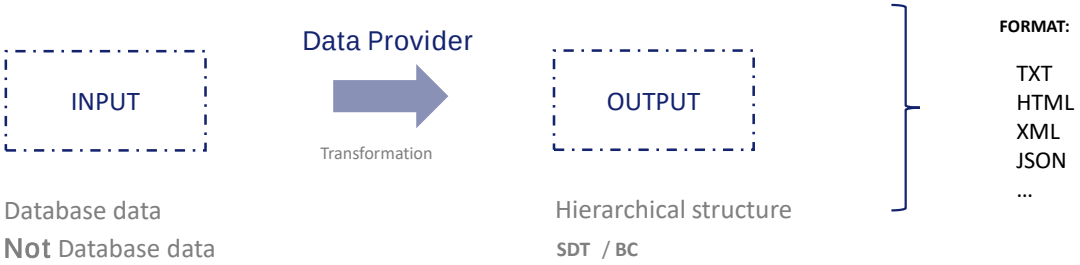
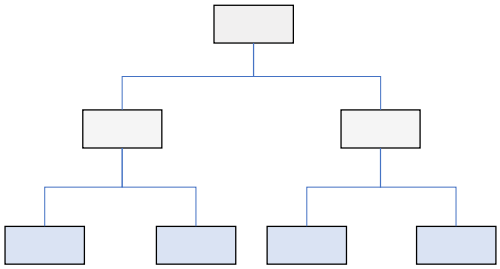
Data Providers

Language and Some Examples

GeneXus™

Data Provider





The purpose of Data Providers is to obtain hierarchical information so that those who need it can do something with it later.

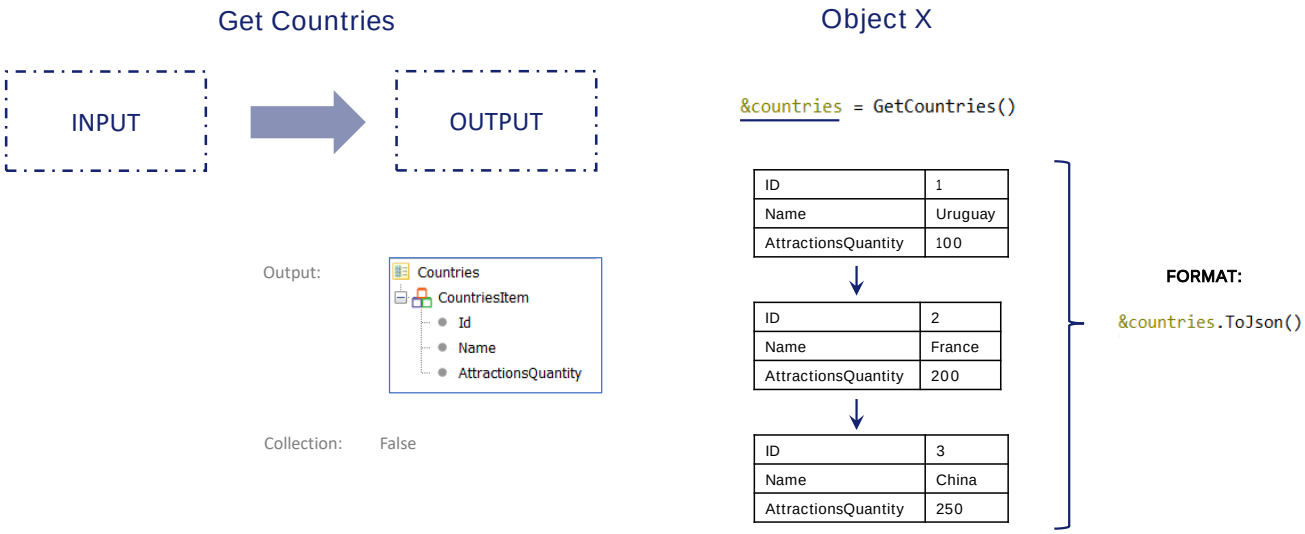
Remember that in Data Providers the focus is placed on the output language: a hierarchical structure indicates how that output is designed. This is why we speak of a process of transforming the input data into this structured output. This data that may or may not be from the database.

The way to represent hierarchical structures in GeneXus is through the SDT object, along with the possibility of defining collections. Of course, a Business Component can be thought of structurally as an SDT. That is why in the Output property of the Data Provider we can specify both an SDT and a Business Component. Also, there is a Collection property to indicate whether the output will be a collection of that indicated data type, or if it will be a single item.

Therefore, a Data Provider will always return a hierarchy to the caller (be it an SDT, a collection of SDTs, a Business Component, or a collection of Business Components).

Whoever invokes it, therefore, is responsible for doing what he/she needs to do with that hierarchical information. For example, convert it into another format for representing hierarchical data, such as XML or JSON which are useful formats for interacting with third parties.

Data Provider



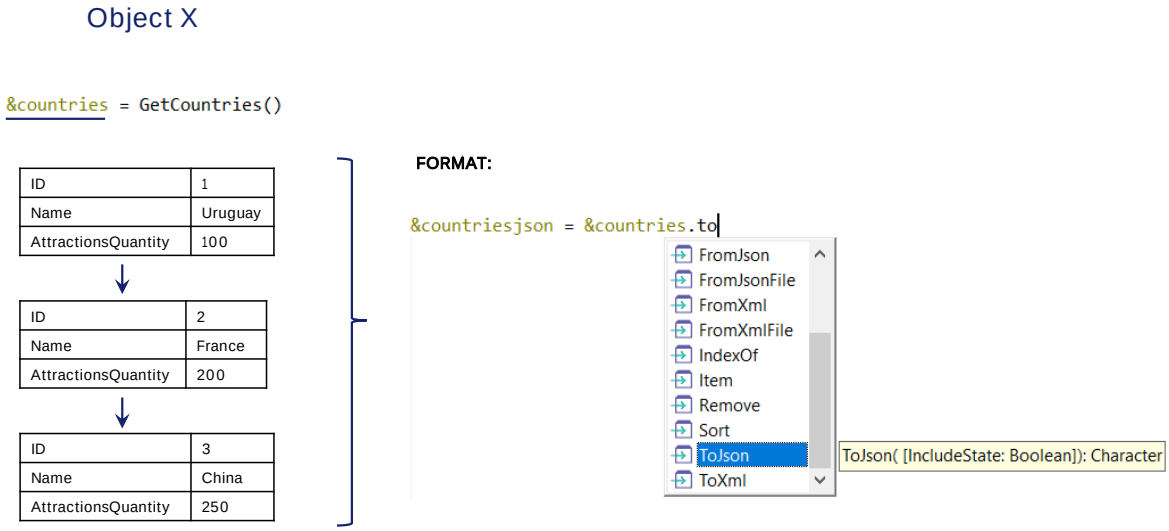
In this example, there is a Data Provider called GetCountries, which will return the data type that we have called Countries. As you can see, it will be a collection of simple SDTs, each of which will take the name CountriesItem.

In the Data Provider properties we will have:

The Output property, with the SDT object named Countries.
And the Collection property set to False, because we don't want a collection of Countries; if it were set to True it would be a collection of collections.

In this example, the Data Provider caller is assigning its result to the countries variable of the same data type as the output. This result will be a specific collection in memory, with specific values, those calculated within that Data Provider.

Then, the program that is working with that variable can do anything with it, for example, convert its content to JSON format.

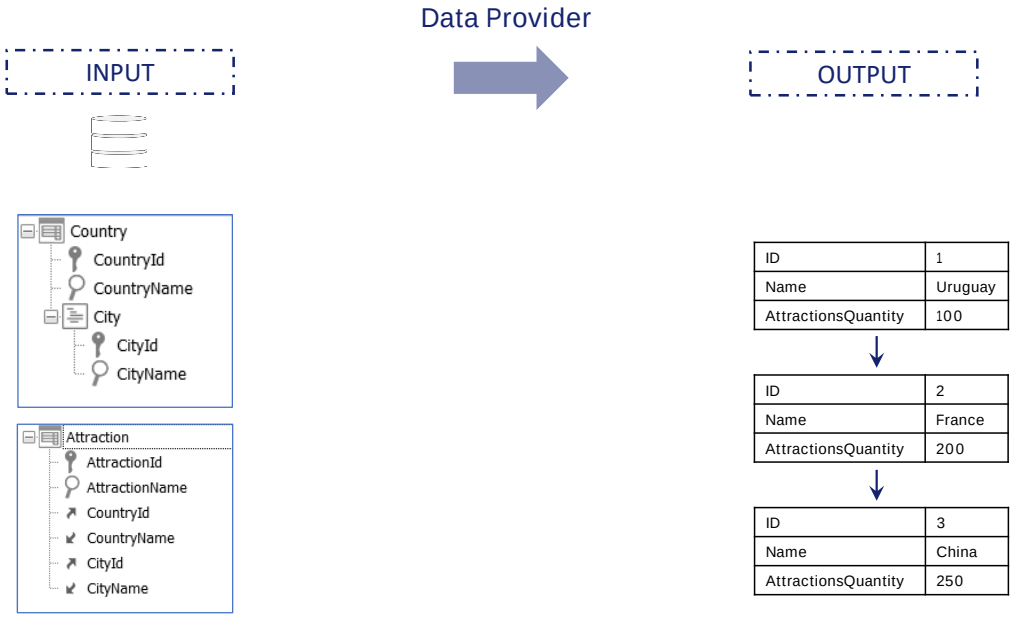


Here, GeneXus offers different conversion methods between SDTs and some of those other formats.

If a new format for representing structured information becomes available in the future, the Data Provider will remain unchanged. GeneXus will implement the conversion to that format, and we will only have to use it.

We can convert from SDT to another format, and vice versa: from that other format to SDT.

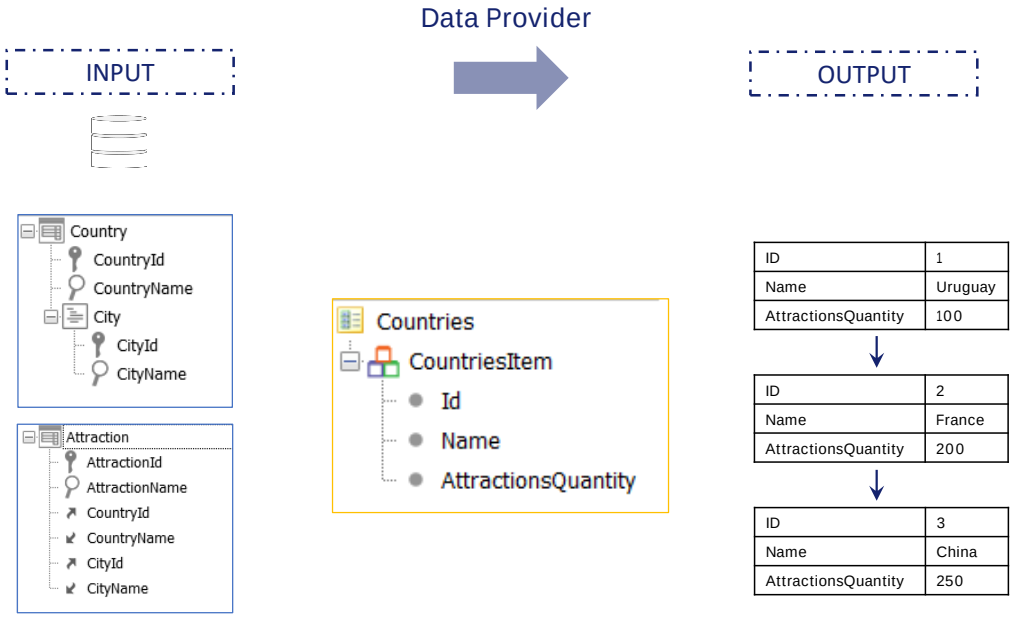
This no longer has to do with the Data Provider itself, but with the structured data types.
The country collection could have been obtained with a procedure instead of a Data Provider, and the conversion part would be identical.



Let's see this example. Let's suppose that, in the context of an application for a travel agency, we need to display on screen a ranking of countries, ordered from highest to lowest by the number of tourist attractions offered by each one.

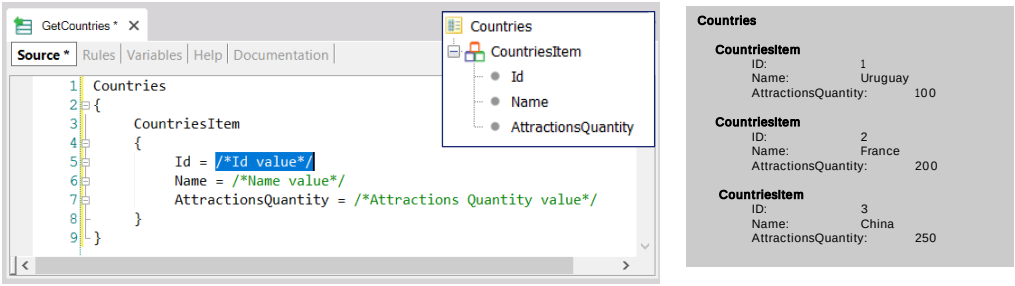
In our reality we have the Country and Attraction transactions with the following attributes.
A simple way to achieve this is to declare a Data Provider that returns a collection of countries where for each one, in addition to its name and identifier, its number of attractions is added. And then process that collection in reverse order by that amount.

As we said, the Data Provider language focuses on the output, the elements are calculated from the point of view of the hierarchy that will be the result.

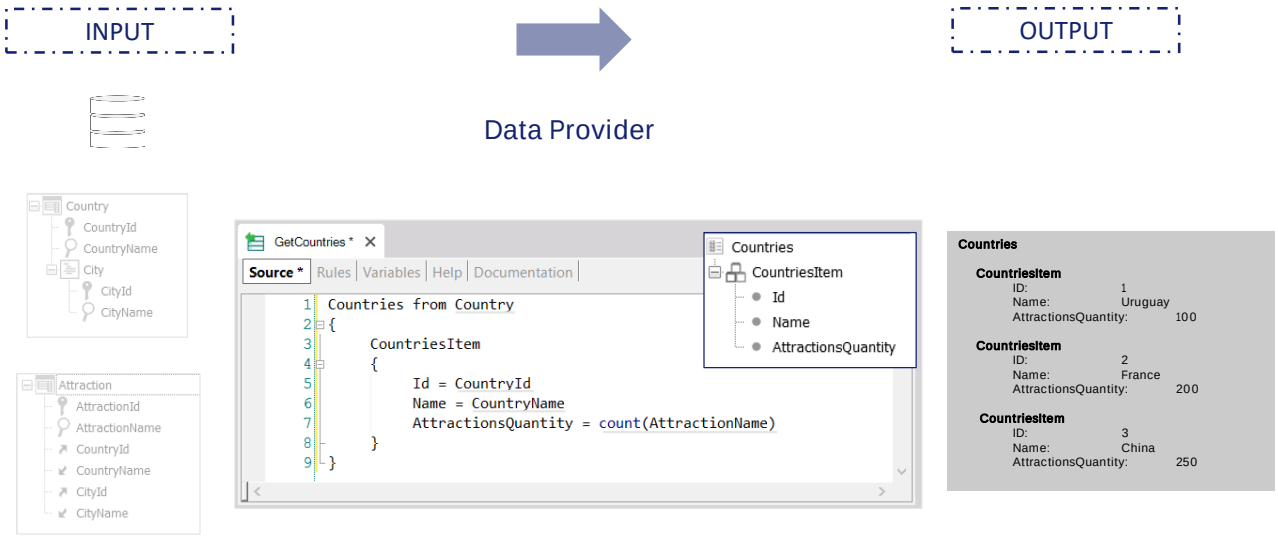


To represent this example, we create the following data structure that will later be returned by the Data Provider. Next, we must load this SDT object into the Data Provider's Source.

Data Provider



By dragging to it the SDT that will be the output of the Data Provider, the structure to be loaded is displayed. We can clearly see how its language is oriented towards the output statement.



Here we see the Input of our Data Provider, that is, where the data is being taken from. There is a specified base transaction, attributes and an inline formula. Clearly data is being taken from the database, to convert it into the required hierarchical data.

INPUT

OUTPUT

Data Provider

GetCountries

Source | Rules | Variables | Help | Documentation

```
1  Countries
2  {
3      CountryiesItem
4      {
5          Id = 1
6          Name = "Uruguay"
7          AttractionQuantity = 100
8      }
9      CountryiesItem
10     {
11         Id = 2
12         Name = "France"
13         AttractionQuantity = 200
14     }
15     CountryiesItem
16     {
17         Id = 3
18         Name = "China"
19         AttractionQuantity = 250
20     }
21 }
```

Countries

CountriesItem
ID: 1
Name: Uruguay
AttractionsQuantity: 100

CountriesItem
ID: 2
Name: France
AttractionsQuantity: 200

CountriesItem
ID: 3
Name: China
AttractionsQuantity: 250

The same result would have been obtained if the data had been statically loaded into the Source, i.e. if the input was not taken from the database but manually coded. As we can see, since there is no base transaction or attributes, GeneXus will not bring information from the database.

GetCountries

Source | Rules | Variables | Help | Documentation

```
Countries
{
    CountriesItem
    {
        Id = 1
        Name = "Uruguay"
        AttractionQuantity = 100
    }
    CountriesItem
    {
        Id = 2
        Name = "France"
        AttractionQuantity = 200
    }
    CountriesItem
    {
        Id = 3
        Name = "China"
        AttractionQuantity = 250
    }
}
```

GetCountries

Source | Rules | Variables | Help | Documentation

```
Countries from Country
{
    CountriesItem
    {
        Id = CountryId
        Name = CountryName
        AttractionQuantity = count(AttractionName)
    }
}
```

MIXED INPUT

GetCountries *

Source * | Rules | Variables | Help | Documentation

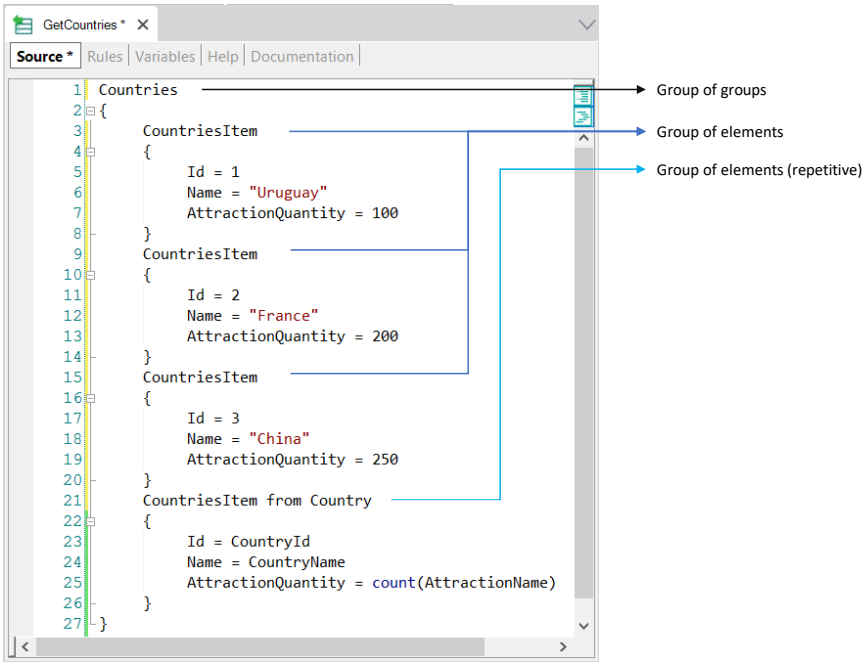
```
Countries
{
    CountriesItem
    {
        Id = 1
        Name = "Uruguay"
        AttractionQuantity = 100
    }
    CountriesItem
    {
        Id = 2
        Name = "France"
        AttractionQuantity = 200
    }
    CountriesItem
    {
        Id = 3
        Name = "China"
        AttractionQuantity = 250
    }
    CountriesItem from Country
    {
        Id = CountryId
        Name = CountryName
        AttractionQuantity = count(AttractionName)
    }
}
```

It is also possible to have a mixed input: one part is statically coded and another part is taken from the database.

In this example, we see in the Data Provider's Source that the output will show three items from the statically loaded collection, and then N more items loaded from the database from the Country table records.

Note that the from clause had to be moved so that it applies to the final CountriesItem subgroup and not to all of them. As we saw, this static part, manually coded by us, does not take records from the database.

- Groups
- Elements
- Variables




```
Countries
{
    CountriesItem
    {
        Id = 1
        Name = "Uruguay"
        AttractionQuantity = 100
    }
    CountriesItem
    {
        Id = 2
        Name = "France"
        AttractionQuantity = 200
    }
    CountriesItem
    {
        Id = 3
        Name = "China"
        AttractionQuantity = 250
    }
    CountriesItem from Country
    {
        Id = CountryId
        Name = CountryName
        AttractionQuantity = count(AttractionName)
    }
}
```

We will now look at the main components of a Data Provider's Source language.

We will have groups, elements and we can also use variables.

The elements are analogous to the members of an SDT. If we think of the hierarchy as a tree, groups are its branches and elements are its leaves. That is, groups are compound elements; they can be made up of other groups and/or elements.

Groups can be static or dynamically loaded; these are called repetitive groups. In this example, the first three groups are static – they are loaded with fixed data–, while the last one is a group that will have an associated base table, and will therefore produce N items in the output, one for each record of the base table considered.

A group with a base table will be equivalent to a For each command.

Data Provider Language

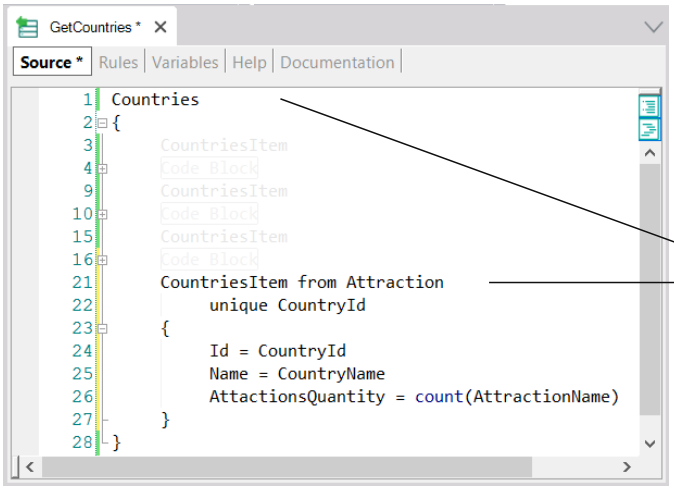
from *BaseTransaction*
[**skip** *expr1*] [**count** *expr2*]
[{[**order**] *order_attributesi* [**when** *condi*]}... | [**order none**] [**when** *condx*]]
[**using** *DataSelectorName*([[*parm1* [,*parm2* [, ...]])]
unique att_1, att_2...,att_n
[{**where** {*conditioni***when** *condi*} |
{*attribute* **IN** *DataSelectorName*([[*parm1* [,*parm2* [, ...]]} }...]

Groups allow specifying a base transaction, although, unlike the For each, here it is specified by preceding the name of the base transaction or level with the word "from."

Note that in this example instead of running through the COUNTRY base table, what we have done is to run through ATTRACTION, using the unique clause so that if there are many attractions in a country, only one is taken into account, and for that one, all the other attractions that have the same country are counted. In this way, only the countries with attractions will be listed in the output.

Everything we've seen about the For Each command is applicable.

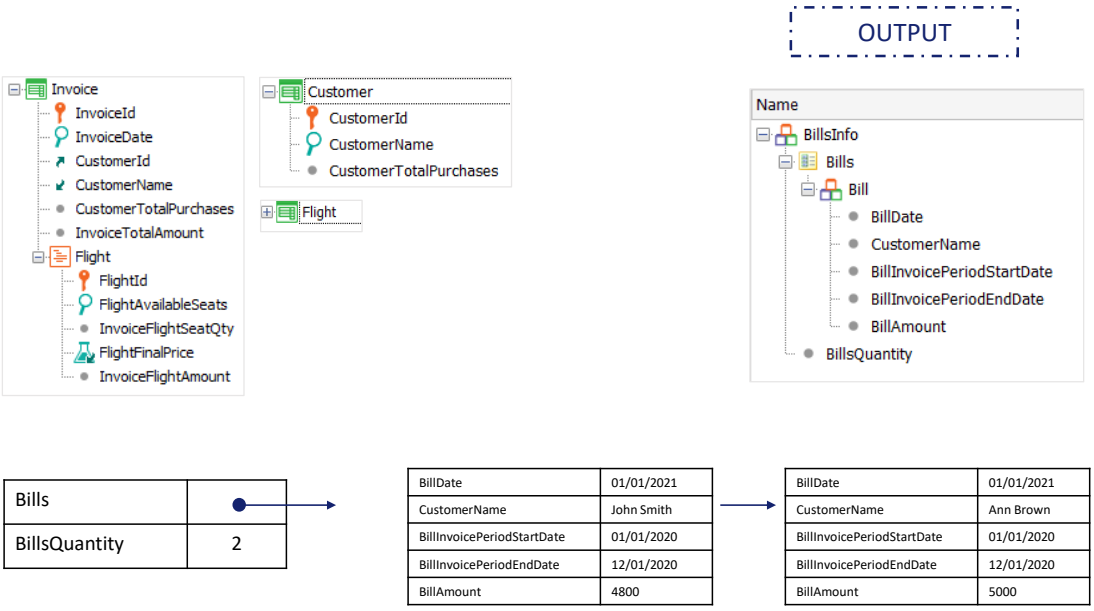
Data Provider Language



from *BaseTransaction*
[**skip** *expr1*] [**count** *expr2*]
[{[**order**] *order_attributesi* [**when** *condi*]}... | [**order none**] [**when** *condx*]]
[**using** *DataSelectorName*([[*parm1* [,*parm2* [, ...]])]
unique att_1, att_2...,att_n
[{**where** {*conditioni***when** *condi*} |
{*attribute* **IN** *DataSelectorName*([[*parm1* [,*parm2* [, ...]]} }...]

If static groups were not required, it would be the same to specify the clauses at the CountriesItem subgroup level or at the parent group level, Countries, CountriesItem collection.

In this case, then, declaring the clauses at the parent group level is equivalent to doing so at the child group level.



Let's see another example.
We have a Data Provider that will return as structure an SDT, which has a collection of Bills and a Quantity element.

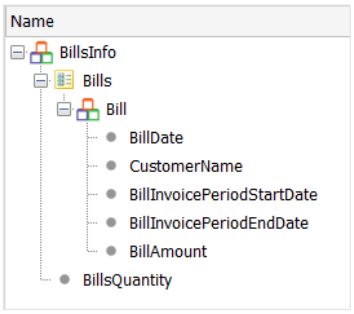
In our application we have the Invoice transaction, which has two levels and the following attributes. The stand-alone Flight transaction, and the Customer transaction.

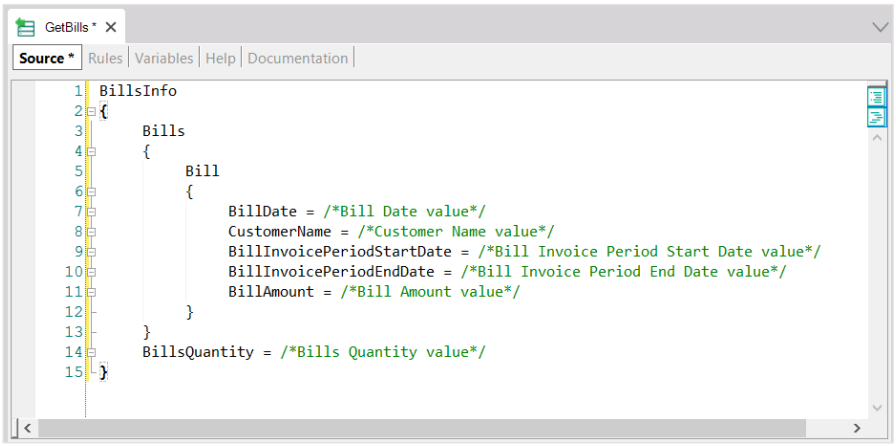
What we want is that from the invoices that have been generated for each customer between two given dates, a payment receipt is generated, for the total of all those invoices. Between these two billing dates, it may happen that not all customers have receipts to be generated, as they may not have been billed in that date range. In the structure to be returned, it is necessary to know how many receipts are obtained from the calculation, and for this we have the BillsQuantity member.

If in the date range 01/01/2020 to 12/01/2020 there are only invoices for the customers John Smith and Ann Brown, and the output will be as represented in the image: an SDT variable with two members: one of Collection type and the other of Numeric type. The collection will have two items, as shown.

In other words, the DataProvider will return a structure with two elements: the collection of receipts on one hand, and the number of items in that collection on the other.

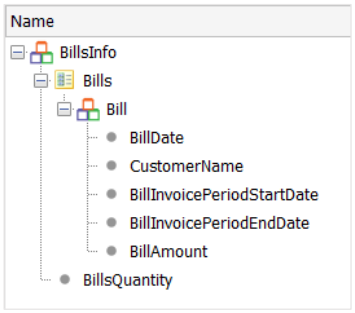
Output: BillsInfo
Collection: False



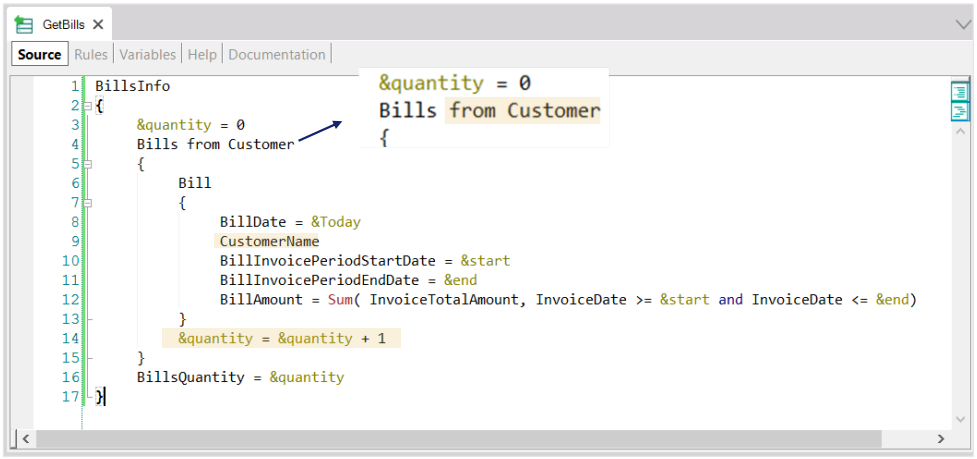


If we drag the SDT to the Data Provider Source, we will see how it is initialized, where the Collection property will be left with its default value "False." This is what we want, because we are not returning a collection of BillsInfo, but only one element of that type which will contain, among other things, a collection of Bills.

Output: BillsInfo
Collection: False



parm(in: &start, in: &end);



We program the parm rule to receive in a parameter the range of billing dates.

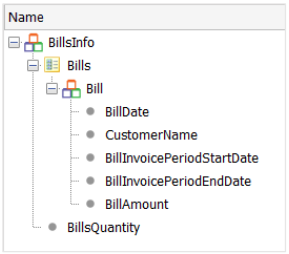
And then for the Bills group, which represents the collection, we specify a base transaction.

Why do we place CustomerName without assigning a value to it? Because its name is the same as the CustomerName attribute and the Customer table is being navigated. So, it is possible to use this abbreviated notation. It is the equivalent to writing: CustomerName equals CustomerName, where the one on the left is the SDT member and the one on the right is the Customer table attribute.

Note that the use of the variables is the same as in a For Each command.

There is a problem: in this case a Bill item will be returned in the output even for customers who do not have invoices in the range received in a parameter. How can this be avoided?

Output: BillsInfo
Collection: False



parm(in: &start, in: &end);

GetBills

Source | Rules | Variables | Help | Documentation

```
1  BillsInfo
2  {
3      &quantity = 0
4      Bills from Invoice
5      unique CustomerId
6      where InvoiceDate >= &start and InvoiceDate <= &end
7      {
8          Bill
9          {
10             BillDate = &Today
11             CustomerName
12             BillInvoicePeriodStartDate = &start
13             BillInvoicePeriodEndDate = &end
14             BillAmount = Sum( InvoiceTotalAmount, InvoiceDate >= &start and InvoiceDate <= &end)
15         }
16         &quantity = &quantity + 1
17     }
18     BillsQuantity = &quantity
19 }
```

One way is to change the base transaction to Invoice and keep the Invoice records without repeating the CustomerId and with dates in the desired range.

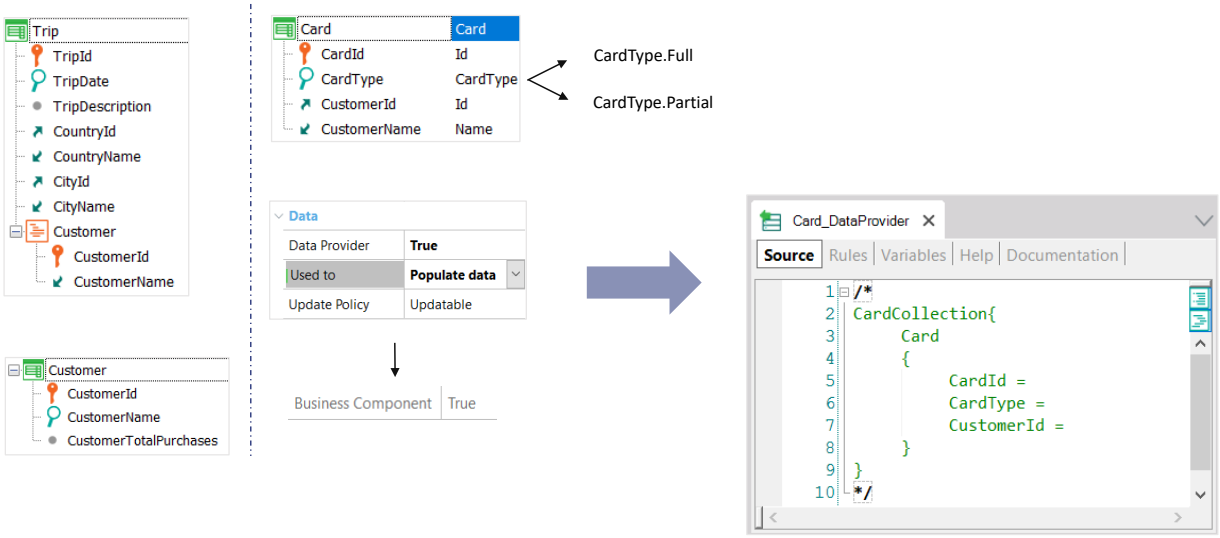
Then in the internal Sum we will count the totals of all the customer's invoices that are in the same date range. As we have been saying, it is identical to the logic of the For each.

Basic components

- Groups
- Elements or Members
- Variables

```
BillsInfo
{
    Bills from
    {
        Bill
        {
            BillDate = &Today
            CustomerName
            BillInvoicePeriodStartDate = &start
            BillInvoicePeriodEndDate = &end
            BillAmount = Sum( InvoiceTotalAmount,
        }
        &quantity = &quantity + 1
    }
    BillsQuantity = &quantity
```

Here see can see, once again, the basic components of a Data Provider's language.



Let's see another example. We want to populate with data the table associated with a new transaction named Card, using its associated Data Provider and data from other tables in the database.

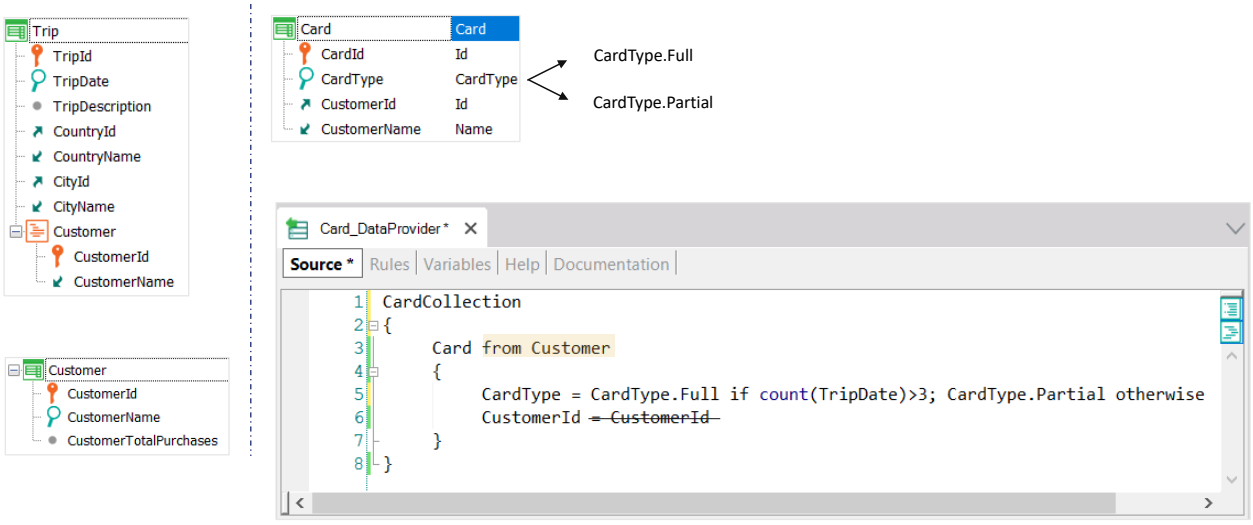
We have the Customer transaction to record the customers of the travel agency and Trip to record each trip or tour offered by the agency in a given city. There is a sublevel with the customers registered for the trip.

Suppose that the travel agency decides that all customers who have booked more than 3 trips will receive a special card of "Full Services" type, which will allow them to enjoy all services free of charge. And if they have booked less than 3 trips, they will receive the "Partial Services" type card.

Each card has an autonumbering identifier, a customer and a card type, for which an enumerated domain has been defined that supports only the values "Full" or "Partial."

Now, we want the table associated with the Card transaction to be initialized with the correct information, so we turn on the Data Provider property and leave the value of Populate data set to "Used to." In this way, it will automatically create the DataProvider that is displayed, and will turn on the Business Component property, to insert the cards that are returned by that Data Provider when it is automatically executed in the first run.

How is the Data Provider Source declared?



We will create a Business Component Card in the collection for each customer. That's why a Customer base transaction is specified for the Card group. We know, therefore, that it is a group with a base table.

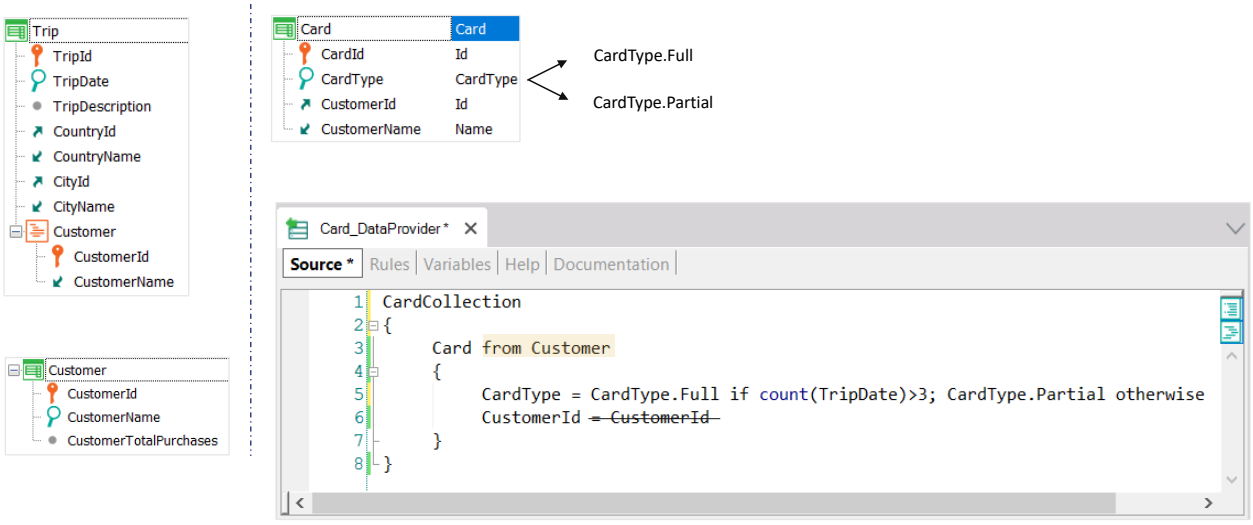
We remove the CardId element from the Card group since the ID domain of the CardId attribute of the transaction is autonumbered.

Next, note how the value of the CardType element is loaded using a conditional inline formula. It will take the value of the enumerated CardType.Full as long as the result of running the inline formula count(TripDate) is greater than 3; otherwise, it will be assigned the value CardType.Partial.

This formula will count the records of the Trip table, filtering by CustomerId.

Then, the CustomerId element, which will correspond to the Business Component, is assigned the value of the CustomerId attribute of the base table of the Customer group. We can use the abbreviated notation and remove the assignment.
Here is an example where the Data Provider returns a collection of Business Components whose data is obtained from another table.

Once again, it is identical to the case of a For each command.



Crearemos un Business Component Card en la colección por cada cliente. Por eso al grupo Card le especificamos transacción base Customer. Sabemos, por tanto, que es un grupo con tabla base.

Quitamos el elemento CardId del grupo Card dado que el dominio Id del atributo CardId de la transacción es autonumerado.

Luego, observemos cómo cargamos el valor del elemento CardType utilizando una fórmula inline condicional. Asumirá el valor del enumerado CardType.Full siempre y cuando el resultado de ejecutar la fórmula inline count(TripDate) sea mayor que 3; de lo contrario se le asignará el valor CardType.Partial.

Esa fórmula irá a contar los registros de la tabla Trip, filtrando por CustomerId.

Y luego al elemento CustomerId, que corresponderá al Business Component, se le asigna el valor del atributo CustomerId de la tabla base del grupo Customer. Podemos utilizar la notación abreviada y quitar la asignación.
Aquí, por tanto, vemos un ejemplo donde el Data Provider devuelve una colección de Business components cuyos datos se obtienen de otra tabla.

Es idéntico, una vez más, al caso de un For each.

GeneXus™

training.genexus.com
wiki.genexus.com