Table of Contents

1. Set up the relationalZoo OR table

I am going to create a table relationalZoo that is a regular table. I am naming the columns with a pattern that starts zoo_ so that we can distinguish them from the xml tags. (Please create this table and follow along- that way you can try more experiments and have a model that works when it comes time to experiment.)

Demo 01: Create and populate relationalZoo

```
Drop table relationalZoo ;

create table relationalZoo  (
  zoo_ID     number(5) primary key
, zoo_type   varchar2(25)
, zoo_name   varchar2(25)
, zoo_price  number(7,2)
, zoo_dob    date
);
```

Demo 02: This is a version of an insert statement that is useful for entering several rows of data with one statement.

```
insert into relationalZoo
      select  51, 'giraffe',  'Dora',    2000, '25-SEP-2015' from dual
union select  52, 'elephant', 'Elmer',   5000, '15-FEB-2000' from dual
union select  53, 'pig',      'Wilbur',  75.5, '18-FEB-2014' from dual
union select  54, 'elephant', 'Arnold',  4000, '28-OCT-2004' from dual
union select  55, 'eland',    'Elvira',  2000, '07-FEB-2014' from dual
union select  61, 'dragon',   'Susan',   null, '29-FEB-1988' from dual
union select  62, 'dragon',    null,     2000,  null         from dual
union select  63, 'dragon',   'Minerva', 2000, '29-FEB-1952' from dual
;
```

We are going to generate xml data from this table.

Do the following set commands to make the output easier to read.

```
set long 999999
set pagesize 999
set feedback 1
```

2. Generating Elements and Attributes

We want to take object-relational data and produce XML instances from that data. It is likely that you have a lot of object-relational data and may need to use it only occasionally as XML. (Oracle refers to its database now as ObjectRelational databases.)

3. Using xmlElement

We will start with the xmlElement method; this generates an xml element.

3.1. Demos

Demo 03: First a few demos; I am going to generate one xml element from one row in the table.

```
select xmlElement (an_name, zoo_Name) as "myXML"
from relationalZoo
where zoo_id = 55;
```

```
myXML
------------------------------
<AN_NAME>Elvira</AN_NAME>

1 row selected.
```

What we can see here is that we gave the xmlElement function two arguments; the first was used to create the element name and was placed inside the start and end tags- which were correctly created with a start and an end tag. The second argument is the name of a column in the table and that supplied the content for the element.

We filtered for a single row and we got one row of well-formed XML returned in the result set. Note that we used this function inside a regular SQL Select statement and took the data from a regular Oracle table with varchar, number and date columns.

Demo 04: Since XML is case sensitive, we might need the element name to be in lower case. In that case, we quote it- that is the standard Oracle approach to things like column headers- use quotes to preserve case. This needs the double quote character.

```
select xmlElement ("an_name", zoo_Name) as "myXML"
from relationalZoo
where zoo_id = 55;
```

```
myXML
-------------------------
<an_name>Elvira</an_name>
1 row selected.
```

Demo 05: selecting a row with a null; empty elements are still well formed xml

```
select xmlElement ("an_name", zoo_Name) as "myXML"
from relationalZoo
where zoo_id = 62;
```

```
myXML
-------------------------
<an_name></an_name>
1 row selected.
```

Demo 06: filtering for multiple rows

```
select xmlElement ("an_name", zoo_Name) as "myXML"
from relationalZoo
where zoo_id in (51,52,53);
```

```
myXML
-------------------------
<an_name>Dora</an_name>
<an_name>Elmer</an_name>
<an_name>Wilbur</an_name>
3 rows selected.
```

But we no longer have a single well-formed XML document. We do not have a single root element. You could describe this situation as missing a root element or as have a multi-rooted document. Each row taken by itself is well-formed XML.

These elements are sometimes called XML fragments because we may be thinking of them as part of an XML document.

3.2. xmlElement syntax

xmlElement is a function that returns an xmltype instance. The output is not a string but SQL*Plus can display it as a string.

xmlElement has a required parameter which is the identifier for the element name. This becomes the root name of the xmltype instance returned. Since XML documents are case specific, you will usually quote that argument to control the case of the element name.

There is an optional second parameter for attributes which we will get to shortly.

The last parameter, which can repeat, is an expression that makes up the element contents.

3.3. subelements

What we want to get is an xml fragment like this for each selected row from the table.

```
<animal>
  <an_id>51</an_id>
  <an_name>Dora</an_name>
  <an_cost>2000</an_cost>
</animal>
```

Thinking about this, we have the child elements (which look like something returned by xmlElement) nested inside the <animal> element which we can get from xmlElement.

Demo 07: This is the SQL to use- where the "third" parameter is repeated three times using an xmlElement function return value.

```
select xmlElement (
       "animal"
       ,  xmlElement ("an_id",   zoo_Id)
       ,  xmlElement ("an_name", zoo_Name)
       ,  xmlElement ("an_cost", zoo_price)
       ) as "myXML"
from relationalZoo
where zoo_ID = 51
;
```

I am reformatting this by hand, adding line breaks, to make the result more obvious

```
myXML
-----------------------------------------------------------------------------
<animal>
  <an_id>51</an_id>
  <an_name>Dora</an_name>
  <an_cost>2000</an_cost>
</animal>
1 row selected.
```

Demo 08: filtering for multiple rows – we get multiple xml fragments- this has three rows since there were 3 dragons in the relationalZoo table.

```
select xmlElement (
        "animal"
      ,  xmlElement ("an_id",   zoo_Id)
      ,  xmlElement ("an_name", zoo_Name)
      ,  xmlElement ("an_cost", zoo_price)
      ) as "myXML"
from  relationalZoo
where zoo_type = 'dragon';
```

```
myXML
--------------------------------------------------------------------------------
<animal><an_id>61</an_id><an_name>Susan</an_name><an_cost></an_cost></animal>
<animal><an_id>62</an_id><an_name></an_name><an_cost>2000</an_cost></animal>
<animal><an_id>63</an_id><an_name>Minerva</an_name><an_cost>2000</an_cost></animal>
3 rows selected.
```

The SQL for this may look complicated mostly because of the nested parentheses. With any nested expression, build it one item at a time. We will see a better function for this in a moment.

4. Using xmlAttributes

Perhaps we were supposed to generate an XML document where the animalID was to be an attribute of the <animal> element rather than a subelement.

For this we can use the function xmlAttributes; this function can be used only inside an xmlElement function call and it has to be the second argument. Remember the third argument can repeat.

Demo 09: one attribute

```
select xmlElement("animal"
, xmlAttributes(zoo_ID as "an_id") ) as "myXML"
from relationalZoo
where zoo_type in ('elephant')
;
```

```
myXML
---------------------------------
<animal an_id="52"></animal>
<animal an_id="54"></animal>

2 rows selected.
```

Demo 10: two attributes

```
select xmlElement("animal"
, xmlAttributes(zoo_ID as "an_id", zoo_name as "an_name") ) as "myXML"
from relationalZoo
where zoo_type in ('elephant')
;
```

```
myXML
```

```
--------------------------------
<animal an_id="52" an_name="Elmer"></animal>
<animal an_id="54" an_name="Arnold"></animal>
```

The syntax for xmlAttributes uses the syntax (*columnName* as "*attribute_name*").

We can generate several attributes by including multiple arguments in the call to xmlAttributes. This differs from xmlElement where we repeated the function call for each subelement

Quote the attribute name to preserve the case. If you omit the alias clause, then the column name is used as the attribute and it defaults to uppercase. If a column is null, the corresponding attribute is not generated.

Demo 11: You may want to generate XML which includes both attributes and subelements.

```
select    xmlElement("animal"
          , xmlAttributes(zoo_ID as "an_id")
          , xmlElement ("an_name", zoo_Name)
          , xmlElement ("an_price", zoo_price)
          ) as "myXML"
from relationalZoo ;
```

```
myXML
--------------------------------------------------------------------------------
<animal an_id="51"><an_name>Dora</an_name><an_price>2000</an_price></animal>
<animal an_id="52"><an_name>Elmer</an_name><an_price>5000</an_price></animal>
<animal an_id="53"><an_name>Wilbur</an_name><an_price>75.5</an_price></animal>
<animal an_id="54"><an_name>Arnold</an_name><an_price>4000</an_price></animal>
<animal an_id="55"><an_name>Elvira</an_name><an_price>2000</an_price></animal>
<animal an_id="61"><an_name>Susan</an_name><an_price></an_price></animal>
<animal an_id="62"><an_name></an_name><an_price>2000</an_price></animal>
<animal an_id="63"><an_name>Minerva</an_name><an_price>2000</an_price></animal>

8 rows selected.
```

Demo 12: More subelements.- this is picking up all of the columns from relationalZoo.

```
select
  xmlElement("animal"
,   xmlAttributes(zoo_ID as "an_id")
,   xmlElement ("an_name",   zoo_Name)
,   xmlElement ("an_price",  zoo_price)
,   xmlElement ("an_type",   zoo_type)
,   xmlElement ("an_dob",    zoo_dob)
    ) as "myXML"
from relationalZoo
;
```

```
myXML
------------------------------------------------------------------------------------------
<animal
an_id="51"><an_name>Dora</an_name><an_price>2000</an_price><an_type>giraffe</an_type><an_d
ob>2002-09-25</an_dob></animal>

<animal
an_id="52"><an_name>Elmer</an_name><an_price>5000</an_price><an_type>elephant</an_type><an
_dob>2000-02-15</an_dob></animal>
```

```
<animal
an_id="53"><an_name>Wilbur</an_name><an_price>75.5</an_price><an_type>pig</an_type><an_dob
>2004-02-18</an_dob></animal>

<animal
an_id="54"><an_name>Arnold</an_name><an_price>4000</an_price><an_type>elephant</an_type><a
n_dob>2004-10-28</an_dob></animal>

<animal
an_id="55"><an_name>Elvira</an_name><an_price>2000</an_price><an_type>eland</an_type><an_d
ob>2006-02-07</an_dob></animal>

<animal
an_id="61"><an_name>Susan</an_name><an_price></an_price><an_type>dragon</an_type><an_dob>1
988-02-29</an_dob></animal>

<animal
an_id="62"><an_name></an_name><an_price>2000</an_price><an_type>dragon</an_type><an_dob></
an_dob></animal>

<animal
an_id="63"><an_name>Minerva</an_name><an_price>2000</an_price><an_type>dragon</an_type><an
_dob>1952-02-29</an_dob></animal>

8 rows selected.
```

5. Using xmlForest

The method xmlForest is a function that produces multiple XML sub-elements; these are considered forests of XML elements. This simply makes it easier to do the above queries. There are a few subtle differences in the output of xmlForest and xmlElement.

With xmlForest, we specify the column expressions to be used as sub-elements along with an element name. This uses the "as *column alias*" syntax.

Notice the difference between xmlForest and xmlElement when we have null values (compare rows 61-63)

Demo 13: xmlForest.

```
select    xmlElement("animal"
          , xmlAttributes(zoo_ID as "an_id")
          , xmlForest (
               zoo_Name   as "an_name"
             , zoo_price  as "an_price"
             , zoo_type   as "an_type"
             , zoo_dob    as "an_dob"
             )
          ) as "myXML"
from relationalZoo
;
```

```
myXML
--------------------------------------------------------------------------------
<animal an_id="51"><an_name>Dora</an_name><an_price>2000</an_price><an_type>gira
ffe</an_type><an_dob>2002-09-25</an_dob></animal>

<animal an_id="52"><an_name>Elmer</an_name><an_price>5000</an_price><an_type>ele
phant</an_type><an_dob>2000-02-15</an_dob></animal>
```

```
<animal an_id="53"><an_name>Wilbur</an_name><an_price>75.5</an_price><an_type>pi
g</an_type><an_dob>2004-02-18</an_dob></animal>

<animal an_id="54"><an_name>Arnold</an_name><an_price>4000</an_price><an_type>el
ephant</an_type><an_dob>2004-10-28</an_dob></animal>

<animal an_id="55"><an_name>Elvira</an_name><an_price>2000</an_price><an_type>el
and</an_type><an_dob>2006-02-07</an_dob></animal>

<animal an_id="61"><an_name>Susan</an_name><an_type>dragon</an_type><an_dob>1988
-02-29</an_dob></animal>

<animal an_id="62"><an_price>2000</an_price><an_type>dragon</an_type></animal>

<animal an_id="63"><an_name>Minerva</an_name><an_price>2000</an_price><an_type>d
ragon</an_type><an_dob>1952-02-29</an_dob></animal>


8 rows selected.
```

this is row 61 from the previous query

```
<animal
an_id="61"><an_name>Susan</an_name><an_price></an_price><an_type>dragon</an_type><an_dob>1
988-02-29</an_dob></animal>
```

this is row 61 from the current query- no price subelement

```
<animal an_id="61"><an_name>Susan</an_name><an_type>dragon</an_type><an_dob>1988
-02-29</an_dob></animal>
```

and this is the insert values for that row

```
select  61, 'dragon',   'Susan',   null, '29-FEB-1988' from dual
```

6. Special issues

6.1. Nulls

We have seen that when a value in the table is null, the default behaviour for an attribute is to skip the attribute and the default for a child element is to generate an empty element.

Demo 14: If you want to skip the empty subelements if the column is null, you can use a case expression. I also use an_cost as the tag.

```
select xmlElement (
  "animal"
  ,  xmlElement ("an_id",   zoo_Id)
  ,  case when zoo_name is null then null
          else xmlElement ("an_name", zoo_Name) end
  ,  case when zoo_price is null then null
          else xmlElement ("an_cost", zoo_price) end
  ) as "myXML"
  from relationalZoo
  where zoo_type = 'dragon'
;
```

```
myXML
-------------------------------------------------------------------------------
<animal><an_id>61</an_id><an_name>Susan</an_name></animal>
<animal><an_id>62</an_id><an_cost>2000</an_cost></animal>
<animal><an_id>63</an_id><an_name>Minerva</an_name><an_cost>2000</an_cost></animal>

3 rows selected.
```

6.2. DateTime values

Demo 15: The format for date values matches the xml date standard format, not the Oracle default format.

```
select XMLElement("animal"
       , XMLElement("an_id",  zoo_ID)
       , XMLElement("an_dob", zoo_DOB)
       ) as "myXMLOutput "
  from relationalZoo
  where zoo_Type='elephant'
;
```

```
myXMLOutput
------------------------------------------------------------
<animal><an_id>52</an_id><an_dob>2000-02-15</an_dob></animal>
<animal><an_id>54</an_id><an_dob>2004-10-28</an_dob></animal>

2 rows selected.
```

7. xmlQuery and FLWOR (OPTIONAL)

Suppose we have our relationalZoo table and also a relational table of animal food and the type of animal that eats them.

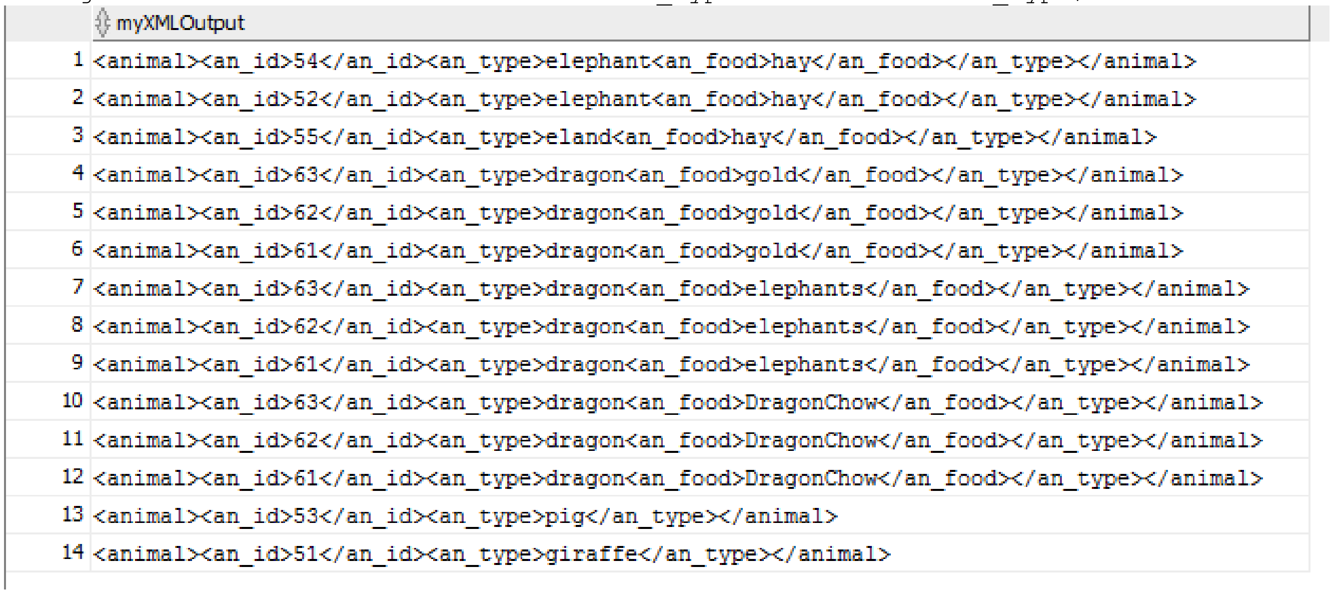
Demo 16: Create and insert and display joined result

```
create table animalFood (an_type varchar2(25), an_food varchar2(25));
insert into animalFood Values ('elephant', 'hay');
insert into animalFood Values ('eland', 'hay');
insert into animalFood Values ('dragon', 'gold');
insert into animalFood Values ('dragon', 'elephants');
insert into animalFood Values ('dragon', 'DragonChow');


select zoo_id, zoo_type,an_food
from relationalZoo
left join animalFood on relationalZoo.zoo_type = animalFood.an_type
;
```

```
    ZOO_ID ZOO_TYPE                  AN_FOOD
---------- ------------------------- -------------------------
        54 elephant                  hay
        52 elephant                  hay
        55 eland                     hay
        63 dragon                    gold
        62 dragon                    gold
        61 dragon                    gold
        63 dragon                    elephants
        62 dragon                    elephants
        61 dragon                     elephants
        63 dragon                     DragonChow
        62 dragon                     DragonChow
        61 dragon                     DragonChow
        53 pig
        51 giraffe

14 rows selected.
```

Demo 17: Working with XMLElement and XMLForrest with the joined tables. This works but notice that any animal which has three foods gets three rows. For example rows 4,7,10 are all for animal id 63.

```
select XMLElement("animal"
       , XMLElement("an_id",  zoo_ID)
       , XMLElement("an_type", zoo_type
              , xmlForest(an_food as "an_food") )
       ) as "myXMLOutput "
from relationalZoo
left join animalFood on relationalZoo.zoo_type = animalFood.an_type;
```



```
    myXMLOutput
 1  <animal><an_id>54</an_id><an_type>elephant<an_food>hay</an_food></an_type></animal>
 2  <animal><an_id>52</an_id><an_type>elephant<an_food>hay</an_food></an_type></animal>
 3  <animal><an_id>55</an_id><an_type>eland<an_food>hay</an_food></an_type></animal>
 4  <animal><an_id>63</an_id><an_type>dragon<an_food>gold</an_food></an_type></animal>
 5  <animal><an_id>62</an_id><an_type>dragon<an_food>gold</an_food></an_type></animal>
 6  <animal><an_id>61</an_id><an_type>dragon<an_food>gold</an_food></an_type></animal>
 7  <animal><an_id>63</an_id><an_type>dragon<an_food>elephants</an_food></an_type></animal>
 8  <animal><an_id>62</an_id><an_type>dragon<an_food>elephants</an_food></an_type></animal>
 9  <animal><an_id>61</an_id><an_type>dragon<an_food>elephants</an_food></an_type></animal>
10  <animal><an_id>63</an_id><an_type>dragon<an_food>DragonChow</an_food></an_type></animal>
11  <animal><an_id>62</an_id><an_type>dragon<an_food>DragonChow</an_food></an_type></animal>
12  <animal><an_id>61</an_id><an_type>dragon<an_food>DragonChow</an_food></an_type></animal>
13  <animal><an_id>53</an_id><an_type>pig</an_type></animal>
14  <animal><an_id>51</an_id><an_type>giraffe</an_type></animal>
```

Demo 18: This is something called a FLWOR query. I will discuss it very briefly at the end of the document. This is a very powerful(although exasperating at times) of working with data to get XML.

```
select xmlQuery (
 '
  <animal_list>
  {
   for $zooanimal in ora:view ("RELATIONALZOO" )
   let $anName  :=  $zooanimal /ROW/ZOO_NAME/text()
     , $zooType :=  $zooanimal /ROW/ZOO_TYPE/text()
   order by $anName
   return
     <animal name="{$anName }">
     <type>{$zooType}</type>
        <menu>
        {for $food in ora:view ("ANIMALFOOD" )
        let $anFood  :=  $food /ROW/AN_FOOD/text()
          , $anType  :=  $food /ROW/AN_TYPE/text()
        where $zooType = $anType
        return
         <food>{$anFood}</food>
        }
        </menu>
     </animal>
  }
  </animal_list>
'
  returning content).extract('/*') as "myXML"
  from dual
;
```

This is the result. This is a single row which contains an XML document. Notice that Susan the dragon only shows up once and her menu includes three foods. I have reformatted this in the next section.

```
myXML
-------------------------------------------------------------------------------

<animal_list><animal name="Arnold"><type>elephant</type><menu><food>hay</food></

menu></animal><animal name="Dora"><type>giraffe</type><menu></menu></animal><ani

mal name="Elmer"><type>elephant</type><menu><food>hay</food></menu></animal><ani

mal name="Elvira"><type>eland</type><menu><food>hay</food></menu></animal><anima

l name="Minerva"><type>dragon</type><menu><food>gold</food><food>elephants</food

><food>DragonChow</food></menu></animal><animal name="Susan"><type>dragon</type>

<menu><food>gold</food><food>elephants</food><food>DragonChow</food></menu></ani

mal><animal name="Wilbur"><type>pig</type><menu></menu></animal><animal name="">

<type>dragon</type><menu><food>gold</food><food>elephants</food><food>DragonChow
```

```
</food></menu></animal></animal_list>


1 row selected.
```

This is the same result set formatted by adding line breaks and whitespace to make the structure clearer. Now we can see that Susan gets one element and the food items are nested within that element. Nicer.

```
   <animal name="Susan">
    <type>dragon</type>
     <menu>
       <food>gold</food>
       <food>elephants</food>
       <food>DragonChow</food>
     </menu>
   </animal>
```

```
myXML
------------------------------------------------------------------------------
<animal_list>
   <animal name="Arnold">
     <type>elephant</type>
     <menu>
       <food>hay</food>
     </menu>
   </animal>
   <animal name="Dora">
     <type>giraffe</type>
     <menu></menu>
   </animal>
   <animal name="Elmer">
     <type>elephant</type>
     <menu>
       <food>hay</food>
     </menu>
   </animal>
   <animal name="Elvira">
     <type>eland</type>
     <menu>
       <food>hay</food>
     </menu>
   </animal>
   <animal name="Minerva">
   <type>dragon</type>
     <menu>
       <food>gold</food>
       <food>elephants</food>
       <food>DragonChow</food>
     </menu>
   </animal>
   <animal name="Susan">
    <type>dragon</type>
     <menu>
```

```
        <food>gold</food>
        <food>elephants</food>
        <food>DragonChow</food>
      </menu>
    </animal>
<animal name="Wilbur">
    <type>pig</type>
      <menu></menu>
    </animal>
    <animal name="">
      <type>dragon</type>
      <menu>
        <food>gold</food>
        <food>elephants</food>
        <food>DragonChow</food>
      </menu>
    </animal>
</animal_list>


1 row selected.
```

Extremely brief discussion of this.

1. `xmlQuery` is an SQL function.
2. `ora:view` is an Oracle function that transforms the data in the table into a series of XML document fragments. The argument to ora:view is a string literal which is the name of a table or view. Each xml document fragment is a <ROW> element for a row in the table.
3. We get back a single row for the table; the row is made up of a sequence of <ROW> elements.
4. The words `for` and `return` inside the quotes are XQuery keywords and must be in lower case letters.
5. The `for` clause loops over the XML document fragments provided by the `ora:view` function.
6. `$v_row` is a variable that gets one XML document fragment at a time.
7. The `return` clause returns each input element
8. The {…} syntax represents a loop

You may have seen this type of loop in another programming language:
For each crt in my_collection Loop . . . End loop
For each element in my_array Loop . . . End Loop
for rec_data in cur_data Loop . . . End Loop

This uses a different syntax but we have the idea that we have a collection of data items and we will process them one at a time. We get a variable to refer to each data item, one after the other.

```
select xmlQuery (             <<< this is an XQuery function
 ' <<< start of the XML structure
  <animal_list>                <<< this starts to build the xml result.

  {     <<<<< starts the first loop
   for $zooanimal in ora:view ("RELATIONALZOO" ) <<< this is a for loop
   let $anName  :=  $zooanimal /ROW/ZOO_NAME/text()    <<< let does
                                                  assignments to variables- use $name
                             this is getting the zoo_name from the relationalZoo table via the view
     , $zooType := $zooanimal /ROW/ZOO_TYPE/text() <<< same for animal type
   order by $anName
   return  <<< this is what we build in XML from those values
     <animal name="{$anName }">    <<< attribute
     <type>{$zooType}</type>       <<< subelement
        <menu>
        {for $food in ora:view ("ANIMALFOOD" ) <<< start of inner loop
         let $anFood  :=  $food /ROW/AN_FOOD/text()
           , $anType  :=  $food /ROW/AN_TYPE/text()
         where $zooType = $anType  <<< this joins the "tables/loops"
         return
           <food>{$anFood}</food>  <<< build more of the XML
         }    <<< end of inner loop
        </menu>
     </animal>
  }       <<< ends the first loop
  </animal_list>
'    <<< end of the XML structure
  returning content).extract('/*') as "myXML"
  from dual
;
```

Aren't you glad the flwor part was optional. That is

F for

L let

W where

O order by

R return