The associate array has three dimensions. The outer dimension is related to the dogs tag from the XML structure. Although there is only one "row" for dogs, a loop is still needed to move into the next array (dog). The foreach loop penetrates the dogs array and provides access to the dog array (which was created from the dog tags in the XML file). $dogs will contain the number of the dog row currently used. $dogs_value will contain the contents of the row (an array with the values in dog_name, dog_weight, dog_color, and dog_breed).

To move through each row (array) contained in the dog array, the method uses a for loop. The conditional statement ($J < count($dogs_value) -1) uses the count method to determine the size of the dog array. The count method returns the size of the array, not the last position of the array. Thus the loop count must be less than (<) the size returned from count. One is subtracted from this value. As stated in Chapter 5, any row after a deleted row must be moved up one from its current position. The last position of the array will no longer be needed, which reduces the number of loops needed by one.

In the Chapter 5 example, a for loop was used to pull each column from the row and place it in the row above. With an associate array, you use a foreach loop. The $column parameter contains the column name ($J contains the row number) to place the values in the columns into the proper locations. After the values in the rows have been moved, the last position of the array is removed using the PHP **unset** method.

Similar logic can be used in almost any program language. However, PHP associate arrays allow index numbers to be skipped. Unlike other languages which would place a NULL value in a missing index, PHP associate arrays just skip the actual index. Thus any array could have indexes of 0, 1, 2, 4, 5 and a foreach loop would properly loop through the array. With this in mind we can greatly simplify the previous example delete example to only contain one line of code, just the unset command shown. The unset command would remove the index passed into the method from the dog_array. Any for loop using the dog_array would still properly loop through the array. The example code contained in the demo website provides this ability.

You can also make a few adjustments to the displayRecords method from Chapter 5 to return any record(s) that a calling program (such as the Dog class) requests.

```
function readRecords($recordNumber) {
if($recordNumber === "ALL") {
                return $this->dogs_array["dog"];
}
else {
                return $this->dogs_array["dog"][$recordNumber];
        } }
```

As you can see the readRecords method is more simplistic that the displayRecords method. All formatting of the results of this method are left to the calling program (if needed). Remember that displaying and formatting of output occurs in the interface tier, not the data tier (or business rules tier).

This method allows the calling program to request all records or a specific record. In either case it returns an array with either one row (the specific record requested) or all rows. When all rows are returned, the top array (representing the rows XML tag) is removed to keep the number of dimensions (two) the same for either selection.

The insertRecords method accepts an associate array with the subscript names previously mentioned. However, in order to allow for dependency injection and flexibility in the tag names for the XML file, the calling program does not need not know the tag names until an instance of the dog_data class is created. This can be accomplished by using the readRecords method to pull the first record and then have the calling program examine the subscript names returned from that record.