Microsoft .NET User Guide for IBM
SPSS Statistics
Before using this information and the product it supports, read the information in "Notices" on page 37.

Product Information
This edition applies to version 23, release 0, modification 0 of IBM SPSS Statistics and to all subsequent releases and modifications until otherwise indicated in new editions.
Chapter 1. Getting Started

The IBM® SPSS® Statistics - Integration Plug-in for Microsoft .NET enables an application developer to create .NET applications that can invoke and control the IBM SPSS Statistics backend through the XD API. The plug-in includes two .NET assemblies, both of which are added to the Global Assembly Cache (GAC) during installation of the plug-in.

- **SPSS.BackendAPI.dll.** A .NET assembly which loads spssxd_p.dll and makes the XD API available from .NET.
- **SPSS.BackendAPI.Controller.dll.** A .NET assembly that wraps the low level XD API functions exposed to .NET by SPSS.BackendAPI.dll. It provides functions that invoke IBM SPSS Statistics, execute syntax commands, provide access to the data, and allow for the creation of custom pivot tables. This is the library intended for use by application developers.

*Note:* The IBM SPSS Statistics - Integration Plug-in for Microsoft .NET is strictly for use by an external .NET application that needs to access the IBM SPSS Statistics backend processor. It is not for use from within IBM SPSS Statistics.

You invoke the IBM SPSS Statistics backend with the Processor class from the SPSS.BackendAPI.Controller library. The constructor for this class is overloaded and provides two approaches for enabling the class instance to locate the IBM SPSS Statistics backend library spssxd_p.dll. spssxd_p.dll is located in the IBM SPSS Statistics install directory and should not be moved.

- **Processor().** Creates an instance of the Processor class and uses the location of IBM SPSS Statistics 23 stored in the Windows registry.
- **Processor(SpssXdPath).** Creates an instance of the Processor class and looks for spssxd_p.dll in the location specified by SpssXdPath.

To help get you started with writing applications for the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET, a sample Visual Basic Class (shown below) is provided that gives an example of creating an instance of the Processor class. How you access the sample class and use the plug-in in your development environment depends on whether you have Visual Studio.

**Visual Studio users.** The sample class is installed as the Visual Studio item template SPSS_Statistics-.NET_Integration_Plug-In230Class (accessible from the Add New Item choice on the Visual Studio Project menu). Adding the item template to a project creates references to the copies of SPSS.BackendAPI.dll and SPSS.BackendAPI.Controller.dll stored in the GAC so there is no need to manually add references to these libraries during development.

**Users without Visual Studio.** The sample class is available for download from Example VB.NET Class on the SPSS Community. To use the sample class you’ll need to manually create references to SPSS.BackendAPI.dll and SPSS.BackendAPI.Controller.dll. Both files can be found in the NET directory under the directory where IBM SPSS Statistics 23 is installed--for example, C:\Program Files\IBM\SPSS\Statistics\23\NET.

```vbnet
Imports SPSS.BackendAPI.Controller
Imports Microsoft.Win32

Public Class SPSS_Statistics_
    Sub PlugIn()
        Dim SPSS_Processor As Processor
        Dim SpssXdPath As String = "" 'Optional: Add your SPSS Statistics location
        If Not SpssXdPath Is String.Empty Then
            SPSS_Processor = New Processor(SpssXdPath)
        Else
            SPSS_Processor = New Processor()
        End If
        'Todo: Add your code here
    End Sub
```
The sample class includes the following items that should always be present in an application that uses the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET:

- An Imports SPSS.BackendAPI.Controller statement. It is up to you to decide how to deploy SPSS.BackendAPI.Controller.dll and SPSS.BackendAPI.dll with your application. See the topic Chapter 9, “Deploying Your Application,” on page 35 for more information.

- A statement to instantiate the Processor class, either with or without an argument specifying the location of spssxd_p.dll.

In addition to instantiating the Processor class, you must start the IBM SPSS Statistics backend before utilizing most of the available APIs. You can do this directly by calling the StartSPSS method, as in SPSS_Processor.StartSPSS(). Note that the IBM SPSS Statistics backend is automatically started when you call the Submit method, so an explicit call to StartSPSS may not be necessary.

- A call to the StopSPSS method of the Processor instance. This method should be called before exiting your application to ensure that all temporary files are removed.

Example

The following is a simple example of using the sample class to create a dataset in IBM SPSS Statistics, compute descriptive statistics and generate output.

```csharp
Imports SPSS.BackendAPI.Controller
Imports Microsoft.Win32
Public Class SPSS_Statistics_
    Sub PlugIn()
        Dim SPSS_Processor As Processor
        SPSS_Processor = New Processor()
        SPSS_Processor.StartSPSS("c:\temp\test.txt")
        Dim cmdLines As System.Array = New String() {
            "DATA LIST FREE /salary (F).",
            "BEGIN DATA",
            "21450",
            "30000",
            "57000",
            "END DATA.",
            "DESCRIPTIVES salary."}
        SPSS_Processor.Submit(cmdLines)
    End Sub
End Class
```

- The StartSPSS method is called with an argument specifying the path to a file where any output is written. In this example, output will be generated by the DESCRIPTIVES procedure and written to the file c:\temp\test.txt.

- A string array specifies IBM SPSS Statistics command syntax that creates a dataset and runs the DESCRIPTIVES procedure. The command syntax is submitted to IBM SPSS Statistics using the Submit method. See the topic Chapter 2, “Running IBM SPSS Statistics Commands,” on page 5 for more information.

You can instantiate the sample class and run the example code from the following module:

```csharp
Module Module1
    Sub Main()
        Dim p As New SPSS_Statistics_
        p.PlugIn()
    End Sub
End Module
```

Documentation
Documentation for the SPSS.BackendAPI.Controller library can be found in the NET directory under the directory where IBM SPSS Statistics 23 is installed—for example, C:\Program Files\IBM\SPSS\Statistics\23\NET. For Visual Studio users, the documentation is integrated into the Visual Studio IDE, and available from the help contents and from the Dynamic Help window. The documentation includes simple code examples for each of the available methods.
Chapter 2. Running IBM SPSS Statistics Commands

The Submit method from the Processor class is used to submit syntax commands to IBM SPSS Statistics for processing. It takes a string that resolves to one or more complete syntax commands, or an array of strings that resolves to one or more complete syntax commands. By default, output from syntax commands is written to the standard output stream. You can direct output to an in-memory workspace where it is stored as XML and can then be retrieved using XPath expressions. See the topic Chapter 6, "Retrieving Output from Syntax Commands," on page 19 for more information.

Submitting a Single Command

You submit a single command to IBM SPSS Statistics by providing a string representation of the command as shown in this example. When submitting a single command in this manner the period (.) at the end of the command is optional.

Processor.Submit("GET FILE='c:/data/Employee data.sav'.")  
Processor.Submit("DESCRIPTIVES SALARY.")

- The Submit method is called twice; first to submit a GET command and then to submit a DESCRIPTIVES command.

Submitting Commands Using an Array

You can submit multiple commands as an array of strings where each array element is a string representation of a syntax command. The string for each command must be terminated with a period (.) as shown in this example.

Processor.Submit("GET FILE='c:/data/Employee data.sav'.")  
Dim cmdLines As System.Array = New String()  
{"DESCRIPTIVES SALARY SALBEGIN.","FREQUENCIES EDUC JOBCAT."}  
Processor.Submit(cmdLines)

- The Submit method is called with an array that specifies a DESCRIPTIVES and a FREQUENCIES command.

You can also use the elements of an array to represent parts of a command so that a single array specifies one or more complete syntax commands. When submitting multiple commands in this manner, each command must be terminated with a period (.) as shown in this example.

Processor.Submit("GET FILE='c:/data/Employee data.sav'.")  
Dim cmdLines As System.Array = New String()  
{"OMS /SELECT TABLES ",  
"/IF COMMANDS = ['Descriptives' 'Frequencies'] ",  
"/DESTINATION FORMAT = HTML ",  
"IMAGES = NO OUTFILE = 'c:/temp/stats.html'.","  
"DESCRIPTIVES SALARY SALBEGIN.",  
"FREQUENCIES EDUC JOBCAT.",  
"OMSEND."}  
Processor.Submit(cmdLines)

- The Submit method is called with an array that specifies an OMS command followed by a DESCRIPTIVES command, a FREQUENCIES command, and an OMSEND command. The first four elements of the array are used to specify the OMS command.

Submitting Multiple Commands In a Single String

You can specify multiple commands in a single string using the character sequence "\n" as a delimiter as shown in this example.

Processor.Submit("GET FILE='c:/data/Employee data.sav'.")  
Processor.Submit("DESCRIPTIVES SALARY.\nFREQUENCIES EDUC.")
• A single string is used to submit both a FREQUENCIES command and a DESCRIPTIVES command. IBM SPSS Statistics interprets "\n" as a newline character.

Displaying Command Syntax Generated by the Submit Method

For debugging purposes, it is convenient to see the completed syntax passed to IBM SPSS Statistics by any calls to the Submit method. This is enabled through command syntax with SET PRINTBACK ON MPRINT ON.

```vbnet
Dim varName As String
varName = Processor.GetVariableName(1)
Processor.Submit("FREQUENCIES /VARIABLES=" + varName + ".")
```

The generated command syntax is written to the standard output stream and shows the completed FREQUENCIES command as well as the GET command. In the present example the variable with index value 1 in the dataset has the name gender.

4 M> GET FILE='c:/data/Employee data.sav'.
6 M> FREQUENCIES /VARIABLES=gender.
Chapter 3. Retrieving Dictionary Information

The Processor class provides a number of methods for retrieving dictionary information from the active dataset. Variables are specified by their position in the dataset, starting with 0 for the first variable in file order. This is referred to as the index value of the variable. The following information is available:

- `GetDataFileAttributeNames()`. Names of any datafile attributes for the active dataset.
- `GetDataFileAttributes(attrName)`. The attribute values for the specified datafile attribute.
- `GetMultiResponseSet(mrsetName)`. The details of the specified multiple response set.
- `GetMultiResponseSetNames()`. The names of any multiple response sets for the active dataset.
- `GetSplitVariableNames()`. Returns the names of the split variables, if any, in the active dataset.
- `GetVariableAttributeNames(index)`. Names of any custom variable attributes for the specified variable.
- `GetVariableAttributes(index, attrName)`. The attribute values for a specified attribute of a specified variable.
- `GetVariableCount()`. The number of variables in the active dataset.
- `GetVariableCValueLabel(index)`. The value labels for the specified string variable.
- `GetVariableFormat(index)`. The display format for the specified variable; for example, F8.2.
- `GetVariableLabel(index)`. The variable label for the specified variable.
- `GetVariableMeasurementLevel(index)`. The measurement level for the specified variable: "nominal", "ordinal", "scale", or "unknown".
- `GetVariableMissingValues(index)`. Any user-missing values for the specified variable.
- `GetVariableName(index)`. The variable name for the specified variable.
- `GetVariableNVValueLabel(index)`. The value labels for the specified numeric variable.
- `GetVariableType(index)`. The variable type (numeric or string) for the specified variable.
- `WeightVariable`. Property that returns the name of the weight variable, if any.

Example

Consider the common scenario of running a particular block of command syntax only if a specific variable exists in the dataset. For example, you are processing many datasets containing employee records and want to split them by gender—if a gender variable exists—to obtain separate statistics for the two gender groups. We will assume that if a gender variable exists, it has the name gender, although it may be spelled in upper case or mixed case. The following sample code illustrates the approach:

```vba
Dim name As String
Processor.Submit("GET FILE='c:/data/Employee data.sav'.")
For i As Integer = 0 To Processor.GetVariableCount() - 1
    name = Processor.GetVariableName(i)
    If LCase(name) = "gender" Then
        Dim cmdLines As System.Array = New String() {'"SORT CASES BY " + name + ".", 
                                                     "SPLIT FILE LAYERED BY " + name + "."}
        Processor.Submit(cmdLines)
        Exit For
    End If
Next i
```
Chapter 4. Working With Case Data in the Active Dataset

The IBM SPSS Statistics - Integration Plug-in for Microsoft .NET provides the ability to read case data from the active dataset, create new variables in the active dataset, and append new cases to the active dataset. Three classes are provided: the DataCursor class allows you to read cases from the active dataset, the DataCursorWrite class allows you to add new variables (and their case values) to the active dataset, and the DataCursorAppend class allows you to append new cases to the active dataset.

The following rules apply to the use of data cursors:

- You cannot use the Submit method from the Processor class while a data cursor is open. You must close the cursor first using the Close method. In particular, if you need to save changes made to the active dataset to an external file, then use the Submit method to submit a SAVE command after closing the cursor.
- Only one data cursor can be open at any point in an application. To define a new data cursor, you must first close the previous one.

Note: For users of the 14.0.2 version of the plug-in who are upgrading to the 16.0.1 version, the GetDataCursorInstance method from the Processor class has been deprecated and replaced with the GetReadCursorInstance method from the Processor class.

Reading Case Data

To retrieve case data, you first create an instance of the DataCursor class using the GetReadCursorInstance method from the Processor class. To specify that case data are to be retrieved for all variables in the active dataset, use the form of the method without an argument, as in GetReadCursorInstance().

You can also specify a selected set of variables to retrieve. Variables are specified by their position in the dataset, starting with 0 for the first variable in file order. For example, to create an instance of the DataCursor class for retrieving case data for just the 5th and 3rd variables in file order, use GetReadCursorInstance(indexSet), where indexSet is a 1-dimension array with the elements 5 and 3.

The following rules apply to retrieved values:
- String values are right-padded to the defined width of the string variable.
- System-missing values are always returned as a value of Nothing. By default, user-missing values are returned as a value of Nothing. You can specify that user-missing values be treated as valid with the UserMissingInclude property from the DataCursor class.
- By default, data retrieved from a variable representing a date, or a date and a time, is given as the number of seconds from October 14, 1582. You can specify a set of IBM SPSS Statistics variables with date or datetime formats to convert to Date data type values when reading data from IBM SPSS Statistics. See the example on handling datetime values.
- The DataCursor class honors case filters specified with the FILTER or USE commands.

Retrieving Cases Sequentially

You can retrieve cases one at a time in sequential order using the GetRow method from the DataCursor class. For example:

```vbnet
Processor.Submit("GET FILE='c:/data/demo.sav'.")
Dim cur As DataCursor = Processor.GetReadCursorInstance()
Dim result As System.Array
'First case
```
The case data is returned as a 1-dimensional array where each element corresponds to the case data for a specific variable. When you invoke the `DataCursor` class with `GetReadCursorInstance()`, as in this example, the elements correspond to the variables in file order.

Retrieving All Cases

You can retrieve all cases at once using the `GetAllRows` method from the `DataCursor` class. For example:

```csharp
Processor.Submit("GET FILE='c:/data/demo.sav'.")
Dim cur as DataCursor = Processor.GetReadCursorInstance()
Dim result as System.Array
result = cur.GetAllRows()
cur.Close()
```

The case data is returned as a 2-dimensional array where the first dimension refers to the case number (starting with 0 for the first case) and the second dimension indexes the variables in the active dataset. For example, `result(n,m)` is the value of the mth variable for the nth case. When you invoke the `DataCursor` class with `GetReadCursorInstance()`, as in this example, the elements in the second dimension correspond to the variables in file order.

Missing Data

By default, missing values (user and system) are returned as `Nothing`.

```csharp
Dim cmdLines As System.Array = New String() _
{"DATA LIST LIST (',') /numVar (f) stringVar (a4).", _
"BEGIN DATA", _
"1,a", _
",b", _
"3," _
",d", _
"9,d", _
"END DATA.", _
"MISSING VALUES numVar (9) stringVar (' ')."}
Processor.Submit(cmdLines)
Dim cur As DataCursor = Processor.GetReadCursorInstance()
Dim result as System.Array
result = cur.GetAllRows()
cur.Close()
```

The values of `result` are:

```
1   a
Nothing   b
3   Nothing
Nothing   d
```

You can specify that user-missing values be treated as valid data by setting the `UserMissingInclude` property of the `DataCursor` instance to `True`, as shown in the following reworking of the previous example.

```csharp
Dim cmdLines As System.Array = New String() _
{"DATA LIST LIST (',') /numVar (f) stringVar (a4).", _
"BEGIN DATA", _
"1,a", _
",b", _
"3," _
",d", _
"9,d", _
"END DATA.", _
"MISSING VALUES numVar (9) stringVar (' ')."}
Processor.Submit(cmdLines)
Dim cur As DataCursor = Processor.GetReadCursorInstance()
Dim result as System.Array
cur.UserMissingInclude = True
result = cur.GetAllRows()
cur.Close()
```

The values of `result` are:
Handling Data with Splits

When reading datasets in which split file processing is in effect, you'll need to be aware of the behavior at a split boundary. Detecting split changes is necessary when you're creating custom pivot tables from data with splits and want separate results displayed for each split group (using the SplitChange method from the DataCursor class). The IsEndSplit method, from the DataCursor class, allows you to detect split changes when reading from datasets that have splits.

Dim cmdLines As System.Array = New String() -
("DATA LIST FREE /salary (F) jobcat (F).", _
"BEGIN DATA", _
"21450 1", _
"45000 1", _
"30000 2", _
"30750 2", _
"103750 3", _
"72500 3", _
"57000 3", _
"END DATA.", _
"SPLIT FILE BY jobcat.")
Processor.Submit(cmdLines)
Dim cur As DataCursor = Processor.GetReadCursorInstance()
cur.UserMissingInclude = False
For i As Integer = 1 To cur.CaseCount()
    cur.GetRow()
    If cur.IsEndSplit() Then
        Console.WriteLine("A new split begins at case: {0}", i)
    cur.GetRow()
    End If
Next
cur.Close()

- cur.IsEndSplit() returns a Boolean—true if a split boundary has been crossed, and false otherwise.
  For the sample dataset used in this example, split boundaries are crossed when reading the 3rd and 5th cases.
- The value returned from the GetRow method is Nothing at a split boundary. In the current example, this
  means that Nothing is returned when attempting to read the 3rd and 5th cases. Once a split has been
  detected, you call GetRow again to retrieve the first case of the next split group, as shown in this
  example.
- Although not shown in this example, IsEndSplit also returns true when the end of the dataset has
  been reached. This scenario would occur if you replace the For loop with a While True loop that
  continues reading until the end of the dataset is detected. Although a split boundary and the end of
  the dataset both result in a return value of true from IsEndSplit, the end of the dataset is identified by
  a return value of Nothing from a subsequent call to GetRow.

Handling IBM SPSS Statistics Datetime Values

When retrieving values of IBM SPSS Statistics variables with date or datetime formats, you'll most likely want to convert the values to Date data type values. By default, date or datetime variables are not
converted and are simply returned in the internal representation used by IBM SPSS Statistics (floating
point numbers representing some number of seconds and fractional seconds from an initial date and
time). To convert variables with date or datetime formats to Date data type values, you use the
GetReadCursorInstance(indexSet,dateindexSet) form of the GetReadCursorInstance method.

Dim cmdLines As System.Array = New String() -
("DATA LIST FREE /case (F) value (DATE10).", _
"BEGIN DATA", _
"1 28-OCT-1990", _
"END DATA.")
Processor.Submit(cmdLines)
Dim indexSet() As Integer = {0, 1}
Dim dateindexSet() As Integer = {1}
Dim cur As DataCursor = Processor.GetReadCursorInstance(indexSet, dateindexSet)
Dim result As System.Array
result = cur.GetRow()
cur.Close()

- The first argument to GetReadCursorInstance specifies the set of variables to retrieve and the second argument specifies the set of variables to convert to Date data type values. The values specified for the second argument must be a subset of those for the first argument. Variables are specified by their position in the dataset, starting with 0 for the first variable in file order.

The value of result is:
1 10/28/1990 12:00:00 AM

Creating New Variables in the Active Dataset

To add new variables along with their case values to the active dataset, you first create an instance of the DataCursorWrite class using the GetWriteCursorInstance method from the Processor class.

- All of the methods available with the DataCursor class are also available with the DataCursorWrite class.

- When adding new variables, the CommitDictionary method must be called after the statements defining the new variables and prior to setting case values for those variables. You cannot add new variables to an empty dataset.

- When setting case values for new variables, the CommitCase method must be called for each case that is modified. The GetRow method is used to advance the record pointer by one case, or you can use the GetRows method to advance the record pointer by a specified number of cases.

- Changes to the active dataset do not take effect until the cursor is closed.

- Write mode supports multiple data passes and allows you to add new variables on each pass. In the case of multiple data passes where you need to add variables on a data pass other than the first, you must call the AllocNewVarsBuffer method to allocate the buffer size for the new variables. When used, AllocNewVarsBuffer must be called before reading any data with GetRow, GetRows, or GetAllRows.

- The SetVariableNameAndType and SetOneVariableNameAndType methods, from the DataCursorWrite class, are used to add new variables to the active dataset. The SetValueChar and SetValueNumeric methods are used to set case values.

Example

In this example we create a new string variable and a new numeric variable and populate their case values for the first and third cases in the active dataset. A sample dataset is first created.

Dim cmdLines As System.Array = New String() _
("DATA LIST FREE /case (A5).", _
"BEGIN DATA", _
"case1", _
"case2", _
"case3", _
"END DATA.")
Processor.Submit(cmdLines)
Dim cur As DataCursorWrite = Processor.GetWriteCursorInstance()
Dim varName As String() = {"numvar", "strvar"}
Dim varType As Integer() = {0, 1}
Dim varLabel As String() = {
    "Sample numeric variable", _
    "Sample string variable"
}
cur.SetVariableNameAndType(varName, varType)
cur.SetVariableLabel(varName(0), varLabel(0))
cur.SetVariableLabel(varName(1), varLabel(1))
cur.CommitDictionary()
cur.GetRow()
cur.SetValueNumeric(varName(0), 1.0)
cur.SetValueChar(varName(1), "a")
cur.CommitCase()
cur.GetRows(2)
cur.SetValueNumeric(varName(0), 3.0)
cur.SetValueChar(varName(1), "c")
cur.CommitCase()
cur.Close()
• The first argument to the SetVariableNameAndType method is an array of strings that specifies the name of each new variable. The second argument is an array of integers specifying the variable type of each variable. Numeric variables are specified by a value of 0 for the variable type. String variables are specified with a type equal to the defined length of the string (maximum of 32767). In this example, we create a numeric variable named numvar and a string variable of length 1 named strvar.

• After calling SetVariableNameAndType you have the option of specifying variable properties (in addition to the variable type) such as the measurement level, variable label, and missing values. In this example variable labels are specified using the SetVariableLabel method.

• Specifications for new variables must be committed to the cursor's dictionary before case values can be set. This is accomplished by calling the CommitDictionary method, which takes no arguments. The active dataset's dictionary is updated when the cursor is closed.

• To set case values, you first position the record pointer to the desired case using the GetRow or GetRows method. GetRow advances the record pointer by one case and GetRows advances it by a specified number of cases. In this example we set case values for the first and third cases. Note: To set the value for the first case in the dataset you must call GetRow as shown in this example.

• Case values are set using the SetValueNumeric method for numeric variables and the SetValueChar method for string variables. For both methods, the first argument is the variable name and the second argument is the value for the current case. A numeric variable whose value is not specified, or specified as Nothing, is set to the system-missing value. A string variable whose value is not specified, or specified as Nothing, will have a blank value. When setting values of IBM SPSS Statistics date or date/time variables, you can specify a Date data type as the value for the SetValueNumeric method. If you specify an integer value, it will be interpreted as the number of seconds from October 14, 1582.

• The CommitCase method must be called to commit the values for each modified case. Changes to the active dataset take effect when the cursor is closed.

Note: To save the modified active dataset to an external file, use the Submit method (following the Close method) to submit a SAVE command, as in:

Processor.Submit("SAVE OUTFILE='c:/data/mydata.sav'.")

Example: Multiple Data Passes

Sometimes more than one pass of the data is required, as in the following example involving two data passes. The first data pass is used to read the data and compute a summary statistic. The second data pass is used to add a summary variable to the active dataset.

```vbnet
Dim cmdLines As System.Array = New String() {"DATA LIST FREE /var (F).", _
"BEGIN DATA", _
"57000", _
"40200", _
"21450", _
"21900", _
"END DATA."}
Processor.Submit(cmdLines)
Dim cur As DataCursorWrite = Processor.GetWriteCursorInstance()
Dim row As System.Array
Dim total As Integer = 0
cur.AllocNewVariablesBuffer(8)
For i As Integer = 1 To cur.CaseCount()
    row = cur.GetRow()
    total = total + row(0)
Next
Dim meanVal As Double = total / cur.CaseCount()
cur.Reset()
cur.SetOneVariableNameAndType("mean", 0)
cur.CommitDictionary()
For i As Integer = 1 To cur.CaseCount()
    row = cur.GetRow()
    cur.SetValueNumeric("mean", meanVal)
    cur.CommitCase()
Next
cur.Close()
```
Because we'll be adding a new variable on the second data pass, the AllocNewVarsBuffer method is called to allocate the required space. In the current example we're creating a single numeric variable, which requires 8 bytes.

The first For loop is used to read the data and total the case values.

After the data pass, the Reset method must be called prior to defining new variables.

The SetOneVariableNameAndType method is used to add a single new variable. The first argument is the variable name and the second argument is the variable type. In this example, we create a numeric variable named mean. The CommitDictionary method is called to commit this variable to the cursor.

The second data pass (second For loop) is used to set the case values of the new variable.

### Appending New Cases

To append new cases to the active dataset, you first create an instance of the DataCursorAppend class using the GetAppendCursorInstance method from the Processor class. The DataCursorAppend class cannot be used to add new variables or read case data from the active dataset. A dataset must contain at least one variable in order to append cases to it, but it need not contain any cases.

- The CommitCase method must be called for each case that is added.
- The EndChanges method must be called before the cursor is closed.
- Changes to the active dataset do not take effect until the cursor is closed.
- The SetValueChar and SetValueNumeric methods are used to set variable values for new cases.

#### Example

In this example two new cases are appended to the active dataset.

```vbnet
Dim cmdLines As System.Array = New String() _
{'DATA LIST FREE /case (F) value (A1).', _
"BEGIN DATA", _
"1 a", _
"END DATA."}
Processor.Submit(cmdLines)
Dim cur As DataCursorAppend = Processor.GetAppendCursorInstance()
cur.SetValueNumeric("case", 2)
cur.SetValueChar("value", "b")
cur.CommitCase()
cur.SetValueNumeric("case", 3)
cur.SetValueChar("value", "c")
cur.CommitCase()
cur.EndChanges()
cur.Close()
```

For both the SetValueNumeric and SetValueChar methods, the first argument is the variable name, as a string, and the second argument is the value for the current case. A numeric variable whose value is not specified, or specified as Nothing, is set to the system-missing value. A string variable whose value is not specified, or specified as Nothing, will have a blank value. The value will be valid unless you explicitly define the blank value to be missing for that variable. When setting values of IBM SPSS Statistics date or date/time variables, you can specify a Date data type as the value for the SetValueNumeric method. If you specify an integer value, it will be interpreted as the number of seconds from October 14, 1582.

The CommitCase method must be called to commit the values for each new case. Changes to the active dataset take effect when the cursor is closed. When working in append mode, the cursor is ready to accept values for a new case (using SetValueNumeric and SetValueChar) once CommitCase has been called for the previous case.

The EndChanges method signals the end of appending cases and must be called before the cursor is closed or the new cases will be lost.
Chapter 5. Creating and Managing Multiple Datasets

Using a data step, you can create and work concurrently with multiple datasets. Data steps are initiated with the GetDatastepInstance method from the Processor class.

Once a data step has been initiated, you access existing datasets and create new datasets with the DsDataset class. DsDataset objects provide access to the case data and variable information contained in a dataset, allowing you to read from the dataset, add new cases, modify existing cases, add new variables, and modify properties of existing variables.

Limitations
- Within a data step you cannot create a cursor, a pivot table, or a text block, and you cannot call the StartProcedure method or the Submit method from the Processor class.
- You cannot start a data step if there are pending transformations. If you need to access case data in the presence of pending transformations, use a cursor.
- Only one data step can exist at a time.
- An instance of the DsDataset class cannot be used outside of the data step in which it was created.
  
  Note: DsDataset objects can also be created between StartProcedure and EndProcedure and not associated with a data step.
- A new dataset created with the DsDataset class is not set to be the active dataset. To make the dataset the active one, use the SetActive method of the associated Datastep object.
- The DsDataset class does not honor case filters specified with the FILTER or USE commands.

Example: Creating a New Dataset

```vbscript
Dim datastepObj As Datastep = Processor.GetDatastepInstance()
Dim dsObj As New DsDataset("newDataset", True)
Dim varsObj As DsVariableCollection = dsObj.VarList
Dim caseObj As DsCases = dsObj.Cases
Dim varObj As DsVariable
varsObj.Add("numvar", 0)
varsObj.Add("strvar", 1)
varObj = varsObj.Item("numvar")
varObj.Label = "Sample numeric variable"
varObj = varsObj.Item("strvar")
varObj.Label = "Sample string variable"
Dim values As Object() = {1, "a"}
 caseObj.Add(values)
 values(0) = 2
 values(1) = "b"
 caseObj.Add(values)
datastepObj.Dispose()
```

- Once a data step has been initiated, you create a new dataset with the DsDataset class. The first argument specifies the name of the new dataset, and setting the second argument to True specifies that this DsDataset object is a new dataset (you can also create DsDataset objects for existing datasets, allowing you to access and/or modify cases and variables).
- You add variables to a dataset using the Add (or Insert) method of the DsVariableCollection object associated with the dataset. The DsVariableCollection object is accessed from the VarList property of the DsDataset object, as in dsObj.VarList. The arguments to the Add method are the name of the new variable and the variable type (0 for numeric variables, and an integer equal to the defined length of the string for string variables).
- Variable properties, such as the variable label and measurement level, are set through properties of the associated DsVariable object, accessible from the DsVariableCollection object. For example, varsObj.Item("numvar") accesses the DsVariable object associated with the variable numvar.
- You add cases to a dataset using the Add (or Insert) method of the DsCases object associated with the dataset. The DsCases object is accessed from the Cases property of the DsDataset object, as in
The argument to the Add method is an array specifying the case values. Values specified in the array must match the corresponding variable type—a double for a numeric variable, and a string for a string variable.

Note: When setting case values of IBM SPSS Statistics date or date/time variables, you can specify the value as a Date data type. If you specify an integer value, it will be interpreted as the number of seconds from October 14, 1582.

• You end a data step with the Dispose method of the associated Datastep instance.

Example: Saving New Datasets

When creating new datasets that you intend to save, you’ll want to keep track of the dataset names since the save operation is done outside of the associated data step. In this example, a sample dataset containing employee data for three departments is first created. Three new datasets, one for each department, are then created and saved.

```
Dim cmdLines As System.Array
    cmdLines = New String()
        (*DATA LIST FREE /dept (F2) empid (F4) salary (F6).*, _
     *BEGIN DATA*.
        "7 57 57000", _
        "5 23 40200", _
        "3 62 21450", _
        "3 18 21900", _
        "5 21 45000", _
        "5 29 32100", _
        "7 38 36000", _
        "5 42 21900", _
        "7 11 27900", _
        *END DATA*.
     *DATASET NAME saldata.*._
     *SORT CASES BY dept.*.)
    Processor.Submit(cmdLines)
Dim datastepObj As Datastep = Processor.GetDatastepInstance()
Dim dsObj As New DsDataset()
Dim caseObj As DsCases = dsObj.Cases
Dim newdsObj As DsDataset
Dim newvarsObj As DsVariableCollection
Dim newcaseObj As DsCases
Dim dept As Integer
Dim dsNames As New StringDictionary()
    ' Create the new datasets
    newdsObj = New DsDataset(Nothing, True)
    newvarsObj = newdsObj.VarList
    newcaseObj = newdsObj.Cases
dept = caseObj.Item(0, 0)
dsNames.Add(dept, newdsObj.Name)
newvarsObj.Add("dept", 0)
newvarsObj.Add("empid", 0)
newvarsObj.Add("salary", 0)
For Each row As System.Array In caseObj
    If row(0) <> dept Then
        newdsObj = New DsDataset(Nothing, True)
        dept = row(0)
        dsNames.Add(dept, newdsObj.Name)
        newcaseObj = newdsObj.Cases
        newvarsObj = newdsObj.VarList
        newvarsObj.Add("dept", 0)
        newcaseObj.Add("empid", 0)
        newcaseObj.Add("salary", 0)
    End If
    newcaseObj.Add(row)
Next
datastepObj.Dispose()
    ' Save the new datasets
For Each de As DictionaryEntry In dsNames
    cmdLines = New String()
        (*DATASET ACTIVATE " & de.Value & ".*,
         *SAVE OUTFILE="/mydata/saldata_" & de.Key & ".sav*.")
    Processor.Submit(cmdLines)
Next
```

16 Microsoft .NET User Guide for IBM SPSS Statistics
• Instantiating the DsDataset class without arguments, as in DsDataset(), creates a DsDataset object for the active dataset—in this case, the sample dataset.

• The code newdsObj = New DsDataset(Nothing, True) creates a new dataset. The name of the dataset is available from the Name property, as in newdsObj.Name. In this example, the names of the new datasets are stored to the dictionary dsNames.

• To save new datasets created with the DsDataset class, use the SAVE command after calling the Dispose method to end the data step. In this example, DATASET ACTIVATE is used to activate each new dataset, using the dataset names stored in dsNames.

Example: Modifying Case Values

Dim cmdLines As System.Array
  cmdLines = New String()
  {'DATA LIST FREE /id (F2) salary (DOLLAR8) jobcat (F1).', _
  'BEGIN DATA', _
  '1 57000 3', _
  '3 41000 1', _
  '2 21450 1', _
  'END DATA.'}
Processor.Submit(cmdLines)
Dim datasetObj As Datastep = Processor.GetDatastepInstance()
Dim dsObj As New DsDataset()
Dim caseObj As DsCases = dsObj.Cases
For I As Integer = 0 To caseObj.Count - 1
  caseObj.Item(I, 1) = 1.05 * caseObj.Item(I, 1)
Next
datastepObj.Dispose()
caseObj2.Item(I, nvars) = 1
For J As Integer = 0 To nvars - 1
    If caseObj1.Item(I, J) <> caseObj2.Item(I, J) Then
        caseObj2.Item(I, nvars) = 0
        Exit For
    End If
Next
Next
datastepObj.Dispose()

- The two datasets are first sorted by the variable *id* which is common to both datasets.
- *dsObj1* and *dsObj2* are *DsDataset* objects associated with the two datasets *empdata1* and *empdata2* to be compared.
- The new variable *match*, added to *empdata2*, is set to 1 for cases that are identical and 0 otherwise.
Chapter 6. Retrieving Output from Syntax Commands

To retrieve command output, you first route it via the Output Management System (OMS) to an area in memory referred to as the XML workspace where it is stored as an XPath DOM that conforms to the Output XML Schema. Output is retrieved from this workspace with functions that employ XPath expressions.

Constructing the correct XPath expression (IBM SPSS Statistics currently supports XPath 1.0) requires an understanding of the Output XML schema. The output schema spss-output-1.8.xsd is distributed with IBM SPSS Statistics. Documentation is included in the IBM SPSS Statistics Help system. It is also provided with the IBM SPSS Statistics - Programmability SDK (available from the SPSS Community) and accessible from \documentation\OutputSchema\oms_oxml_schema_intro.htm within the IBM SPSS Statistics - Programmability SDK.

Example

In this example, we'll retrieve the mean value of a variable calculated from the Descriptives procedure.

```v
Dim handle, context, xpath As String
'Route output to the XML workspace.
Dim cmdLines As System.Array = New String() _
"GET FILE='c:/data/Employee data.sav'.", _
"OMS SELECT TABLES ", _
""/IF COMMANDS=['Descriptives'] SUBTYPES=['Descriptive Statistics'] ", _
""/DESTINATION FORMAT=OXML XMLWORKSPACE='desc_table' ", _
""/TAG='desc_out'.", _
"DESCRIPTIVES VARIABLES=salary, salbegin, jobtime, prevexp ", _
""/STATISTICS=MEAN.", _
"OMSEND TAG='desc_out'."
Processor.Submit(cmdLines)
'Get output from the XML workspace using XPath.
handle = "desc_table"
context = "//outputTree"
xpath = "//pivotTable[PsSubType='Descriptive Statistics']" + _
"/dimension[Axis='row']" + _
"/category[VarName='salary']" + _
"/dimension[Axis='column']" + _
"/category[Text='Mean']" + _
"/cell[@text"
Dim result() As String = Processor.EvaluateXPath(handle, context, xpath)
Processor.DeleteXPathHandle(handle)
```

• The OMS command is used to direct output from a syntax command to the XML workspace. The XMLWORKSPACE keyword on the DESTINATION subcommand, along with FORMAT=OXML, specifies the XML workspace as the output destination. It is a good practice to use the TAG subcommand, as done here, so as not to interfere with any other OMS requests that may be operating. The identifiers used for the COMMANDS and SUBTYPES keywords on the IF subcommand can be found in the OMS Identifiers dialog box, available from the Utilities menu in IBM SPSS Statistics.

• The XMLWORKSPACE keyword is used to associate a name with this XPath DOM in the workspace. In the current example, output from the DESCRIPTIVES command will be identified with the name desc_table. You can have many XPath DOM's in the XML workspace, each with its own unique name.

• The OMSEND command terminates active OMS commands, causing the output to be written to the specified destination—i.e., in this case, the XML workspace.

• You retrieve values from the XML workspace with the EvaluateXPath method from the Processor class. The method takes an explicit XPath expression, evaluates it against a specified XPath DOM in the XML workspace, and returns the result as a 1-dimension array of string values.

• The first argument to the EvaluateXPath function specifies the XPath DOM to which an XPath expression will be applied. This argument is referred to as the handle name for the XPath DOM and is simply the name given on the XMLWORKSPACE keyword on the associated OMS command. In this case the handle name is desc_table.
The second argument to EvaluateXPath defines the XPath context for the expression and should be set to "/outputTree" for items routed to the XML workspace by the OMS command.

The third argument to EvaluateXPath specifies the remainder of the XPath expression (the context is the first part) and must be quoted. Since XPath expressions almost always contain quoted strings, you'll need to use a different quote type from that used to enclose the expression. For users familiar with XSLT for OXML and accustomed to including a namespace prefix, note that XPath expressions for the EvaluateXPath function should not contain the oms: namespace prefix.

The XPath expression in this example is specified by the variable xpath. It is not the minimal expression needed to select the mean value of interest but is used for illustration purposes and serves to highlight the structure of the XML output.

('//pivotTable[@subType='Descriptive Statistics']
  /dimension[@axis='row']/category[@varName='salary']
  /dimension[@axis='column']/category[@text='Mean']
  /cell/@text selects the textual representation of the cell contents.

When you have finished with a particular output item, it is a good idea to delete it from the XML workspace. This is done with the DeleteXPathHandle method, whose single argument is the name of the handle associated with the item.

If you're familiar with XPath, you might want to convince yourself that the mean value of salary can also be selected with the following simpler XPath expression:

//category[@varName='salary']//category[@text='Mean']//cell/@text

Note: To the extent possible, construct your XPath expressions using language-independent attributes, such as the variable name rather than the variable label. That will help reduce the translation effort if you need to deploy your code in multiple languages. Also consider factoring out language-dependent identifiers, such as the name of a statistic, into constants. You can obtain the current language used for pivot table output with the syntax command SHOW OLANG.

You may also consider using text_eng attributes in place of text attributes in XPath expressions. text_eng attributes are English versions of text attributes and have the same value regardless of the output language. The OATTRS subcommand of the SET command specifies whether text_eng attributes are included in OXML output.

Retrieving Images Associated with an Output XPath DOM

You can retrieve images associated with output routed to the XML workspace. In this example, we'll retrieve a bar chart associated with output from the Frequencies procedure.

```
Dim cmdLines As System.Array = New String() _
("GET FILE='c:/data/Employee data.sav'.", _
"OMS SELECT CHARTS ", _
"/IF COMMANDS=['Frequencies'] ", _
"/DESTINATION FORMAT=OXML IMAGES=YES", _
"/CHARTFORMAT=IMAGE IMAGEROOT='myimages' IMAGEFORMAT=JPG XMLWORKSPACE='demo',", _
"/FREQUENCIES VARIABLES=jobcat", _
" /BARCHART PERCENT", _
" /ORDER=ANALYSIS.", _
"OMSEND."")
Processor.Submit(cmdLines)
handle = "demo"
context = "/outputTree"
xpath = "/command[0]command='Frequencies']" + _
"/chartTitle[0]text='Bar Chart']" + _
"/chart/imageFile"
Dim result() As String = Processor.EvaluateXPath(handle, context, xpath)
Dim imageName As String = result(0)
Dim imageSize As Integer = 0
Dim imageType As String = String.Empty
Dim imageObj As Byte() = Processor.GetImage(handle, imageName, imageSize, imageType)
Dim fs As New FileStream("c:\temp\result.jpg", FileMode.Create, FileAccess.Write)
```
The OMS command routes output from the FREQUENCIES command to an output XPath DOM with the handle name of demo.

To route images along with the OXML output, the IMAGES keyword on the DESTINATION subcommand (of the OMS command) must be set to YES, and the CHARTFORMAT, MODELFORMAT, or TREEFORMAT keyword must be set to IMAGE.

The EvaluateXPath function is used to retrieve the name of the image associated with the bar chart output from the FREQUENCIES command. In the present example, the value returned by EvaluateXPath is a list with a single element, which is then stored to the variable imageName.

The GetImage function retrieves the image, which is then written to an external file.

The first argument to the GetImage function specifies the particular XPath DOM and must be a valid handle name defined by a previous IBM SPSS Statistics OMS command.

The second argument to GetImage is the filename associated with the image in the OXML output—specifically, the value of the imageFile attribute of the chart element associated with the image.

The third argument to GetImage is the amount of memory required for the image and is a value that is returned by the function.

The fourth argument to GetImage is a string specifying the image type and is a value that is returned by the function. The possible values are: “PNG”, “JPG”, “BMP”.

Writing XML Workspace Contents to a File

When writing and debugging XPath expressions, it is often useful to have a sample file that shows the XML structure. This is provided by the GetXmlUtf16 method from the Processor class, as well as by an option on the OMS syntax command. The following program block recreates the XML workspace for the preceding example and writes the XML associated with the handle desc_table to the file c:\temp\descriptives_table.xml.

Note: The form GetXmlUtf16(String) of the GetXmlUtf16 method returns the content associated with a specified handle, without writing the content to a file.
Retrieving a DOM as an XmlDocument Object

As an alternative to the GetXmlUtf16 method for retrieving an entire XPath DOM, you can use the GetXmlObject method to return the contents of an output DOM as an XmlDocument object. This allows you to process the contents using the powerful methods available for .NET XmlDocument objects. The root element of the associated DOM has a default namespace with the URI http://xml.spss.com/spss/oms, so depending on which XmlDocument methods you use, you may have to add this URI and an associated arbitrary prefix to the XmlNamespaceManager. XPath references will then need to include this prefix, as shown in the following code sample that extracts the text attribute from each cell element, for an XPath DOM specified by handle.

```vbnet
dim result as xml.xmldocument = processor.getxmlobject(handle)
dim nsmgr as xmlnamespacemanager = new xmlnamespacemanager(result.nametable)
nsmgr.addnamespace("spss", "http://xml.spss.com/spss/oms")
dim root as xmlelement = result.documentelement
dim nodeList as xmlnodecollection
nodeList = root.selectnodes("/spss:cell/@text", nsmgr)
```


Chapter 7. Running Python and R Programs from .NET

The IBM SPSS Statistics BEGIN PROGRAM command provides the ability to integrate the capabilities of external programming languages with IBM SPSS Statistics command syntax. The supported languages are the Python programming language and R. This enables you to utilize the extensive set of scientific and statistical programming libraries available with the Python language and R to create custom algorithms that you can apply to your data. Results can be written to a dataset or the XML workspace and then retrieved by your .NET program.

Python Programs

Once you have installed the IBM SPSS Statistics - Integration Plug-in for Python, you have full access to the Python programming language by including Python code in a BEGIN PROGRAM PYTHON-END PROGRAM block (within command syntax) and submitting the command syntax with the Submit method from the Processor class. For example:

```csharp
Dim cmdLines As System.Array
cmdLines = New String() _
{"BEGIN PROGRAM PYTHON.", _
"import spss", _
"<Python code>", _
"END PROGRAM."}
Processor.Submit(cmdLines)
```

- The Python statement `import spss` imports the Python modules (installed with the IBM SPSS Statistics - Integration Plug-in for Python) that allow the Python processor to interact with IBM SPSS Statistics.
- Your Python language code follows the `import` statement.
- You can omit the keyword PYTHON on BEGIN PROGRAM—it is the default.

Complete documentation on the functionality available with the IBM SPSS Statistics - Integration Plug-in for Python is available in the SPSS Statistics Help system under Integration Plug-in for Python.

R Programs

Once you have installed the IBM SPSS Statistics - Integration Plug-in for R, you have full access to the R programming language by including R code in a BEGIN PROGRAM R-END PROGRAM block (within command syntax) and submitting the command syntax with the Submit method from the Processor class. For example:

```csharp
Dim cmdLines As System.Array
cmdLines = New String() _
{"BEGIN PROGRAM R.", _
"<R code>", _
"END PROGRAM."}
Processor.Submit(cmdLines)
```

Complete documentation on the functionality available with the IBM SPSS Statistics - Integration Plug-in for R is available in the SPSS Statistics Help system under Integration Plug-in for R.

Inserting Programs From Command Syntax Files

You can use the IBM SPSS Statistics INSERT command to include BEGIN PROGRAM-END PROGRAM blocks contained in command syntax files. This allows you to store your programs as separate code files and include them as needed. For example:

```csharp
Processor.Submit("INSERT FILE='myprograms/program_block.sps'.")
```

The file `myprograms/program_block.sps` would contain a BEGIN PROGRAM block, as in:
BEGIN PROGRAM PYTHON.
import spss
<Python code>
END PROGRAM.
Chapter 8. Creating Custom Output

The IBM SPSS Statistics - Integration Plug-in for Microsoft .NET provides the ability to create output in the form of custom pivot tables and text blocks. Using the Output Management System (OMS), the output can be rendered in a variety of formats such as HTML, text, or XML that conforms to the Output XML Schema (xml.spss.com/spss/oms).

Creating Pivot Tables

The following figure shows the basic structural components of a pivot table. Pivot tables consist of one or more dimensions, each of which can be of the type row, column, or layer. In this example there is one dimension of each type. Each dimension contains a set of categories that label the elements of the dimension—for instance, row labels for a row dimension. A layer dimension allows you to display a separate two dimensional table for each category in the layered dimension—for example, a separate table for each value of minority classification, as shown here. When layers are present the pivot table can be thought of as stacked in layers, with only the top layer visible.

Each cell in the table can be specified by a combination of category values. In the example shown here, the indicated cell is specified by a category value of Male for the Gender dimension, Custodial for the Employment Category dimension, and No for the Minority Classification dimension.

![Figure 1. Pivot table structure](image-url)

The SimplePivotTable Method

Pivot tables are created with the BasePivotTable class. For the common case of creating a pivot table with a single row dimension and a single column dimension, the BasePivotTable class provides the SimplePivotTable method. The arguments to the method provide the dimensions, categories, and cell values. No other methods are necessary in order to create the table structure and populate the cells. See if you require more functionality than the SimplePivotTable method provides.

```vbnet
Dim cmdLines As System.Array = New String() _
{"GET FILE='c:/examples/data/demo.sav'.", _
  "OMS SELECT TABLES", _
  "IF SUBTYPES=['SimpleTableDemo']", _
  "DESTINATION FORMAT=HTML OUTFILE='c:/temp/simpletable.htm'."} _
Processor.Submit(cmdLines) _
Dim title As String = "Sample Pivot Table" _
Dim templateName As String = "SimpleTableDemo" _
Dim table As BasePivotTable = _
  Processor.GetBasePivotTableInstance(title, templateName, Nothing) _
table.SimplePivotTable( _
  "Row Dimension", _
  New Object() {"1", "2"}, _
  "Column Dimension", _
  New String() {"Clerical", "Custodial", "Manager", "Total"} _
  New Object() {166, 0, 10, 176}, _
  New Object() {116, 14, 70, 194}, _
  New Object() {276, 14, 80, 370}) _
```

---

25
The `BasePivotTable` class requires an active dataset, so a dataset is opened with the `GET` syntax command.

In order to produce something other than a textual representation of a pivot table, you route the output with OMS. To route pivot table output to OMS, you include the `TABLES` keyword on the `SELECT` subcommand of the `OMS` command, as in this example. As specified on the `OMS` command, the pivot table in this example will be rendered in HTML and routed to the file `c:/temp/simpletable.htm`.

To create a pivot table, you create an instance of the `BasePivotTable` class with the `GetBasePivotTableInstance` method from the `Processor` class. The first argument to the `GetBasePivotTableInstance` method is a required string that specifies the title that appears with the table.

The second argument to the `GetBasePivotTableInstance` method is a string that specifies the OMS (Output Management System) table subtype for this table. This value is required (even if you don't route the output with OMS), must begin with a letter, and have a maximum of 64 bytes. The table subtype can be used on the `SUBTYPES` keyword of the `OMS` command, as done here, to include this pivot table in the output for a specified set of subtypes.

Note: By creating the pivot table instance within a `StartProcedure-EndProcedure` block, you can associate the pivot table output with a command name, as for pivot table output produced by syntax commands. The command name is the argument to the `StartProcedure` function and can used on the `COMMANDS` keyword of the `OMS` command to include the pivot table in the output for a specified set of commands (along with optionally specifying a subtype as discussed above).

The third argument to the `GetBasePivotTableInstance` method is a string that specifies an optional outline title for the table, or `Nothing`, as in this example. When output is routed to OMS in OXML format, the outline title specifies a heading tag that will contain the output for the pivot table.

Once you've created an instance of the `BasePivotTable` class, you use the `SimplePivotTable` method to create the structure of the table and populate the table cells. The method has four different calling signatures. The signature used in this example makes use of all possible arguments. The arguments used in this example, are in order:

- **rowdim.** A label for the row dimension, given as a string. If empty, the row dimension label is hidden.
- **rowcats.** A 1-dimensional `System.Array` of categories for the row dimension. Labels can be given as numeric values or strings. They can also be specified as a `CellText` object. `CellText` objects allow you to specify that category labels be treated as variable names or variable values, or that cell values be displayed in one of the numeric formats used in pivot tables, such as the format for a mean. When you specify a category as a variable name or variable value, pivot table display options such as display variable labels or display value labels are honored.
  
  Note: The number of rows in the table is equal to the number of elements in `rowcats`, when provided. If `rowcats` is omitted, the number of rows is equal to the number of elements in the argument `cells`.
- **coldim.** A label for the column dimension, given as a string. If empty, the column dimension label is hidden.
• **colcats.** A 1-dimensional System.Array of categories for the column dimension. The array can contain the same types of items as **rowcats** described above.  
*Note:* The number of columns in the table is equal to the number of elements in **colcats**, when provided. If **colcats** is omitted, the number of columns is equal to the length of the first element of **cells**.

• **cells.** This argument specifies the values for the cells of the pivot table, and can be given as a 1- or 2-dimensional System.Array. In the current example, **cells** is given as the 1-dimensional array {"1A", "1B", "2A", "2B"}. It could also have been specified as the 2-dimensional array {{"1A", "1B"}, {"2A", "2B"}}.

  Elements in the pivot table are populated in row-wise fashion from the elements of **cells**. In the current example, the table has two rows and two columns (as specified by the row and column labels), so the first row will consist of the first two elements of **cells** and the second row will consist of the last two elements. When **cells** is 2-dimensional, each 1-dimensional element specifies a row. For example, with **cells** given by {{"1A", "1B"}, {"2A", "2B"}}, the first row is {"1A", "1B"} and the second row is {"2A", "2B"}.

  Cells can be given as numeric values or strings, or CellText objects (as described for **rowcats** above).

• Numeric values specified for cell values, row labels, or column labels, are displayed using the default format for the pivot table. Instances of the BasePivotTable class have an implicit default format of Count, which displays values rounded to the nearest integer. You can change the default format using the SetDefaultFormatSpec method from the BasePivotTable class. You can also override the default format for specific cells or labels by specifying the value as a CellText.Number object and providing a format, as in CellText.Number(22, FormatSpec.GeneralStat).

• The Close method of the BasePivotTable class closes the table object. If the pivot table is not part of a StartProcedure-EndProcedure block, the output will be produced when the last open pivot table or text block is closed. If the pivot table is part of a StartProcedure-EndProcedure block, the output will be produced when EndProcedure is called.

*Note:* If you’re creating a pivot table from data with splits, you’ll probably want separate results displayed for each split group. See the topic "Creating Pivot Tables from Data with Splits" on page 31 for more information.

**General Approach to Creating Pivot Tables**

The BasePivotTable class provides methods for creating pivot tables that can't be created with the SimplePivotTable method. The basic steps for creating a pivot table are:

1. Create an instance of the BasePivotTable class
2. Add dimensions
3. Define categories
4. Set cell values

**Step 1: Adding Dimensions**

You add dimensions to a pivot table with the Append or Insert method.

*Example: Using the Append Method*

```v
Dim table As BasePivotTable = _
  Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim coldim As Dimension = table.Append(Dimension.Place.column, "coldim")
Dim rowdim1 As Dimension = table.Append(Dimension.Place.row, "rowdim-1")
Dim rowdim2 As Dimension = table.Append(Dimension.Place.row, "rowdim-2")
```

• The first argument to the Append method specifies the type of dimension, using a member of the Dimension.Place Enumeration: Dimension.Place.row for a row dimension, Dimension.Place.column for a column dimension, and Dimension.Place.layer for a layer dimension.

• The second argument to Append is a string that specifies the name used to label this dimension in the table.
The order in which the dimensions are appended determines how they are displayed in the table. Each newly appended dimension of a particular type (row, column, or layer) becomes the current innermost dimension in the displayed table. In the example above, rowdim-2 is the innermost row dimension since it is the last one to be appended. Had rowdim-2 been appended first, followed by rowdim-1, rowdim-1 would be the innermost dimension.

Note: Generation of the resulting table requires more code than is shown here.

Example: Using the Insert Method

```vbnet
Dim table As BasePivotTable = Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim rowdim1 As Dimension = table.Append(Dimension.Place.row, "rowdim-1")
Dim rowdim2 As Dimension = table.Append(Dimension.Place.row, "rowdim-2")
Dim rowdim3 As Dimension = table.Insert(2, Dimension.Place.row, "rowdim-3")
Dim coldim As Dimension = table.Append(Dimension.Place.column, "coldim")
```

• The first argument to the Insert method specifies the position within the dimensions of that type (row, column, or layer). The first position has index 1 and defines the innermost dimension of that type in the displayed table. Successive integers specify the next innermost dimension and so on. In the current example, rowdim-3 is inserted at position 2 and rowdim-1 is moved from position 2 to position 3.

• The second argument to Insert specifies the type of dimension, using a member of the Dimension.Place Enumeration: spss.Dimension.Place.row for a row dimension, spss.Dimension.Place.column for a column dimension, and spss.Dimension.Place.layer for a layer dimension.

• The third argument to Insert is a string that specifies the name used to label this dimension in the displayed table.

```
Figure 3. Resulting table structure
```

```
Figure 4. Resulting table structure
```

Note: Generation of the resulting table requires more code than is shown here.

**Step 2: Defining Categories**

Categories for a dimension are defined using the SetCategory method of the associated Dimension object (created with the Append or Insert method).

Example

```vbnet
Dim table As BasePivotTable = Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim coldim As Dimension = table.Append(Dimension.Place.column, "coldim")
Dim rowdim1 As Dimension = table.Append(Dimension.Place.row, "rowdim-1")
Dim rowdim2 As Dimension = table.Append(Dimension.Place.row, "rowdim-2")
coldim.SetCategory(New CellText.NString("C"))
coldim.SetCategory(New CellText.NString("D"))
coldim.SetCategory(New CellText.NString("E"))
rowdim1.SetCategory(New CellText.NString("A1"))
rowdim1.SetCategory(New CellText.NString("A2"))
rowdim2.SetCategory(New CellText.NString("B1"))
rowdim2.SetCategory(New CellText.NString("B2"))
```

• You set categories after you add dimensions, so the SetCategory method calls follow the Append or Insert method calls.
The argument to SetCategory is a single category expressed as a CellText object (one of CellText.Number, CellText.NString, CellText.VarName, or CellText.VarValue). When you specify a category as a variable name or variable value, pivot table display options such as display variable labels or display value labels are honored. In the present example, we use string objects whose single argument is the string specifying the category.

For a given dimension, the order of the categories displayed in the table, is the order in which they are created. For instance, the first column has a category value of "C", the second column has a category value of "D", etc.

<table>
<thead>
<tr>
<th>rowdim.1</th>
<th>rowdim.2</th>
<th>column</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>A2</td>
<td>C</td>
</tr>
<tr>
<td>B1</td>
<td>A2</td>
<td>D</td>
</tr>
</tbody>
</table>

**Figure 5. Resulting table structure**

### Notes
- Generation of the resulting table (shown above) requires more code than is shown here.
- When specifying numeric category values with CellText.Number objects, values will be formatted using the pivot table’s default format, unless a specific format is supplied, as in CellText.Number(22, FormatSpec.GeneralStat). Instances of the BasePivotTable class have an implicit default format of Count, which displays values rounded to the nearest integer. You can change the default format using the SetDefaultFormatSpec method from the BasePivotTable class.

### Step 3: Setting Cell Values
There are two methods for setting cell values: setting values one cell at a time using the SetCell method, or setting entire rows or columns using the SetCellsByRow or SetCellsByColumn method.

#### Example: Using the SetCell Method

This example reproduces the table created in the SimplePivotTable example.

```vba
Dim table As BasePivotTable = Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim rowdim As Dimension = table.Append(Dimension.Place.row, "Row Dimension")
Dim coldim As Dimension = table.Append(Dimension.Place.column, "Column Dimension")
rowdim.SetCategory(New CellText.NString("1"))
rowdim.SetCategory(New CellText.NString("2"))
coldim.SetCategory(New CellText.NString("A"))
coldim.SetCategory(New CellText.NString("B"))
table.SetCell(New Object() {New CellText.NString("1"), New CellText.NString("A")}, New CellText.NString("1A"))
table.SetCell(New Object() {New CellText.NString("1"), New CellText.NString("B")}, New CellText.NString("1B"))
table.SetCell(New Object() {New CellText.NString("2"), New CellText.NString("A")}, New CellText.NString("2A"))
table.SetCell(New Object() {New CellText.NString("2"), New CellText.NString("B")}, New CellText.NString("2B"))
```

- The Append method is used to add a row dimension and then a column dimension to the structure of the table. The table specified in this example has one row dimension and one column dimension.
- The SetCategory method is used to create the four categories needed for the table—two categories for the row dimension and two categories for the column dimension.
- Cell values are set with the SetCell method. The first argument is a 1-dimensional System.Array of categories (one for each dimension) that specifies the cell. The first element specifies a category in the first appended dimension (what we have named "Row Dimension"), the second element specifies a
category in the second appended dimension (what we have named "Column Dimension"), etc. In this example, the array `{New CellText.NString("1"), New CellText.NString("A")}` specifies the cell whose "Row Dimension" category is "1" and "Column Dimension" category is "A".

- The second argument to the `SetCell` method is the cell value. Cell values must be specified as `CellText` objects (one of `CellText.Number`, `CellText.NString`, `CellText.VarName`, or `CellText.VarValue`).

Example: Setting Cell Values by Row or Column

The `SetCellsByRow` and `SetCellsByColumn` methods allow you to set cell values for entire rows or columns with one method call. To illustrate the approach we'll use the `SetCellsByRow` method to reproduce the table created in the SimplePivotTable example. It is a simple matter to rewrite the example to set cells by column.

```v
Dim table As BasePivotTable = _
    Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim rowdim As Dimension = table.Append(Dimension.Place.row, "Row Dimension")
Dim coldim As Dimension = table.Append(Dimension.Place.column, "Column Dimension")
rowdim.SetCategory(New CellText.NString("1"))
rowdim.SetCategory(New CellText.NString("2"))
coldim.SetCategory(New CellText.NString("A"))
coldim.SetCategory(New CellText.NString("B"))
table.SetCellsByRow(New Object() {New CellText.NString("1")}, _
    New Object() {New CellText.NString("1A"), _
    New CellText.NString("1B")})
table.SetCellsByRow(New Object() {New CellText.NString("2")}, _
    New Object() {New CellText.NString("2A"), _
    New CellText.NString("2B")})
```

- The `SetCellsByRow` method is called for each of the two categories in the row dimension.

- The first argument to the `SetCellsByRow` method is a 1-dimensional `System.Array` of categories (one for each row dimension) that specifies a row. Each element of the array must be specified as a `CellText` object (one of `CellText.Number`, `CellText.NString`, `CellText.VarName`, or `CellText.VarValue`).

- The second argument to the `SetCellsByRow` method is a 1-dimensional `System.Array` of cell values that specifies the cells in the row, one element for each column category in the single column dimension. Each element of the array must be specified as a `CellText` object. The first element in the array will populate the first column category, the second will populate the second column category, and so on.

Note

When specifying numeric cell values with `CellText.Number` objects, values will be formatted using the pivot table's default format, unless a specific format is supplied, as in `CellText.Number(22, FormatSpec.GeneralStat)`. Instances of the `BasePivotTable` class have an implicit default format of `Count`, which displays values rounded to the nearest integer. You can change the default format using the `SetDefaultFormatSpec` method from the `BasePivotTable` class.

Generating the Pivot Table Output

The `Close` method of the `BasePivotTable` class closes the table object. If the pivot table is not part of a `StartProcedure-EndProcedure` block, the output will be produced when the last open pivot table or text block is closed. If the pivot table is part of a `StartProcedure-EndProcedure` block, the output will be produced when `EndProcedure` is called.

Using Cell Values in Expressions

Once a cell's value has been set it can be accessed and used to specify the value for another cell. Cell values are stored as a `CellText` object (one of `CellText.Number`, `CellText.NString`, `CellText.VarName`, or `CellText.VarValue`). To use a cell value in an expression, you obtain a string or numeric representation of the value using the `ToString` or `ToNumber` method.

Example: Numeric Representations of Cell Values
Dim table As BasePivotTable = Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim rowdim As Dimension = table.Append(Dimension.Place.row, "Row Dimension"
    rowdim.SetCategory(New CellText.NString("1"))
    rowdim.SetCategory(New CellText.NString("2"))
Dim coldim As Dimension = table.Append(Dimension.Place.column, "Column Dimension"
    coldim.SetCategory(New CellText.NString("A"))
    coldim.SetCategory(New CellText.NString("B"))
Dim cell As Object = table.GetCell(New Object() {New CellText.NString("1"), New CellText.NString("A")})
Dim cellValue As Double = cell.ToNumber()
table.SetCell(New Object() {New CellText.NString("1"), New CellText.NString("B")}, New CellText.Number(2 * cellValue))

- The GetCell method is used to obtain the cell object for a specified cell. The argument is a 1-dimensional System.Array of categories (one for each dimension) that specifies the cell. Each element of the array must be specified as a CellText object. The first element of the array specifies a category in the first appended dimension (what we have named "Row Dimension"), the second element specifies a category in the second appended dimension (what we have named "Column Dimension"), etc.
- In this example, the ToNumber method is used to obtain a numeric representation of the cell object with category values {"1", "A"}. The numeric value of the cell is stored in the variable cellValue and used to specify the value of another cell.
- Character representations of numeric values stored as CellText.NString objects, such as CellText.NString("11"), are converted to a numeric value by the ToNumber method.

Example: String Representations of Cell Values
Dim table As BasePivotTable = Processor.GetBasePivotTableInstance(title, templateName, Nothing)
Dim rowdim As Dimension = table.Append(Dimension.Place.row, "Row Dimension"
    rowdim.SetCategory(New CellText.NString("1"))
    rowdim.SetCategory(New CellText.NString("2"))
Dim coldim As Dimension = table.Append(Dimension.Place.column, "Column Dimension"
    coldim.SetCategory(New CellText.NString("A"))
    coldim.SetCategory(New CellText.NString("B"))
Dim cell As Object = table.GetCell(New Object() {New CellText.NString("1"), New CellText.NString("A")})
Dim cellValue As String = cell.ToString()
table.SetCell(New Object() {New CellText.NString("1"), New CellText.NString("B")}, New CellText.Number(cellValue & "d"))

- The GetCell method is used to obtain the cell object for a specified cell. The argument is a 1-dimensional System.Array of categories (one for each dimension) that specifies the cell. Each element of the array must be specified as a CellText object. The first element of the array specifies a category in the first appended dimension (what we have named "Row Dimension"), the second element specifies a category in the second appended dimension (what we have named "Column Dimension"), etc.
- In this example, the ToString method is used to obtain a string representation of the cell object with category values {"1", "A"}. The string value is stored in the variable cellValue and used to specify the value of another cell.
- Numeric values stored as CellText.Number objects are converted to a string value by the ToString method.

Creating Pivot Tables from Data with Splits
When generating pivot tables from data with splits, you may want to produce separate results for each split group. This is accomplished with the SplitChange method from the DataCursor class (also available with the DataCursorWrite class).
Example

In this example, a split is created and separate averages are calculated for the split groups. Results for different split groups are shown in a single pivot table. In order to understand the example, you will need to be familiar with creating pivot tables using the BasePivotTable class.

```
Dim cmdLines As System.Array = New String() _
{"GET FILE='c:/data/employee data.sav'.", _
"SORT CASES BY GENDER.", _
"SPLIT FILE LAYERED BY GENDER.", _
"OMS SELECT TABLES", _
"/IF SUBTYPES=['SplitChange']", _
"/DESTINATION FORMAT=HTML OUTFILE='c:/temp/splitchange.htm'."}
Processor.Submit(cmdLines)
Dim table As BasePivotTable = Processor.GetBasePivotTableInstance( _
"Table Title", "SplitChange", Nothing)
table.Append(Dimension.Place.row, "Minority Classification")
table.Append(Dimension.Place.column, "coldim", True, False)
Dim cur As DataCursor = Processor.GetReadCursorInstance()
Dim salary As Integer = 0, salarym As Integer = 0
Dim n As Integer = 0, m As Integer = 0
Dim minorityIndex As Integer = 9
Dim salaryIndex As Integer = 5
Dim row As System.Array = cur.GetRow()
cur.SplitChange()
While True
  If cur.IsEndSplit() Then
    If n > 0 Then
      salary = salary / n
      End If
    If m > 0 Then
      salarym = salarym / m
      End If
    'Populate the pivot table with values for the previous split group
    table.SetCell(New Object() _
    {New CellText.NString("No")., New CellText.NString("Average Salary")}, _
    New CellText.Number(salary, FormatSpec.Count))
    table.SetCell(New Object() _
    {New CellText.NString("Yes")., New CellText.NString("Average Salary")}, _
    New CellText.Number(salarym, FormatSpec.Count))
    salary = 0
    salarym = 0
    n = 0
    m = 0
    'Try to get the first case of the next split group
    row = cur.GetRow()
    If Not row Is Nothing Then
      cur.SplitChange()
      Else
        'There are no more cases, so quit
        Exit While
    End If
  ElseIf row(minorityIndex) = 0 Then
    salary += row(salaryIndex)
    m += 1
  ElseIf row(minorityIndex) = 1 Then
    salarym += row(salaryIndex)
    n += 1
  End If
  row = cur.GetRow()
End While
cur.Close()
table.Close()
```

Result

<table>
<thead>
<tr>
<th>Gender</th>
<th>Minority Classification</th>
<th>Average Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Female</td>
<td>No</td>
<td>26701</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>29002</td>
</tr>
<tr>
<td>Male</td>
<td>No</td>
<td>4475</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>33236</td>
</tr>
</tbody>
</table>

Figure 6. Pivot table displaying results for separate split groups
• Command syntax is first submitted to create a split on a gender variable. The LAYERED subcommand on the SPLIT FILE command indicates that results for different split groups are to be displayed in the same table.

• The SplitChange method is called after getting the first case from the active dataset. This is required so that the pivot table output for the first split group is handled correctly.

• Split changes are detected using the IsEndSplit method from the DataCursor class (also available with the DataCursorWrite class). Once a split change is detected, the pivot table is populated with the results from the previous split.

• The value returned from the GetRow method is Nothing at a split boundary. Once a split has been detected, you will need to call GetRow again to retrieve the first case of the new split group, followed by SplitChange.

Note: IsEndSplit returns true when the end of the dataset has been reached. Although a split boundary and the end of the dataset both result in a return value of true from IsEndSplit, the end of the dataset is identified by a return value of Nothing from a subsequent call to GetRow, as shown in this example.

Creating Text Blocks

Text blocks are created with the TextBlock class.

```vbnet
Dim cmdLines As System.Array = New String() {"GET FILE='c:/examples/data/demo.sav'.", _
"OMS SELECT TEXTS", _
"/IF LABELS = ['Text block name']", _
"/DESTINATION FORMAT=HTML OUTFILE='c:/temp/textblock.htm'.")
Processor.Submit(cmdLines)
Dim tb As TextBlock = Processor.GetTextBlockInstance( _
"Text block name", "A single line of text.", Nothing)
tb.Close()
Processor.Submit("OMSEND.")
```

• The TextBlock class requires an active dataset, so a dataset is opened with the GET syntax command.

• In the common case that you’re creating text blocks along with other output such as pivot tables, you’ll probably be routing the output with OMS. To route text block output to OMS, you include the TEXTS keyword on the SELECT subcommand of the OMS command, as in this example. As specified on the OMS command, the text block in this example will be rendered in HTML and routed to the file c:/temp/textblock.htm.

• You create an instance of the TextBlock class with the GetTextBlockInstance method from the Processor class. The first argument to the GetTextBlockInstance method is a required string that specifies the title that appears with the table. The title can be used on the LABELS keyword of the OMS command, as done here, to limit the textual output routed to OMS.

• The second argument to the GetTextBlockInstance method is the content of the text block as a string. Additional lines can be appended using the Append method of the TextBlock instance.

• The third argument to the GetTextBlockInstance method is a string that specifies an optional outline title for the text block, or Nothing, as in this example. When output is routed to OMS in OXML format, the outline title specifies a heading tag that will contain the output for the text block.
Chapter 9. Deploying Your Application

You can deploy a .NET application that you have developed, but your application can only be used with a licensed IBM SPSS Statistics application. For information about licensing or distribution arrangements, please contact IBM Corp. directly.

In order for your application to run properly, the target computer must have:

- An IBM SPSS Statistics application. Support for the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET was introduced in version 14.0.2.
- The .NET Integration assemblies for IBM SPSS Statistics.
- A means for the .NET Integration assemblies to locate spssxd_p.dll.

.NET Integration Assemblies for IBM SPSS Statistics

Copies of these libraries are included with the IBM SPSS Statistics - Integration Plug-in for Microsoft .NET and are located in the NET directory under the directory where IBM SPSS Statistics 23 is installed—for example, C:\Program Files\IBM\SPSS\Statistics\23\NET.

Deploy versions of these libraries that match the version of spssxd_p.dll on the target machine. You can check the version of spssxd_p.dll from the value of SpssdxVersion in spssdxcfg.ini, located in the IBM SPSS Statistics application’s root directory.

Locating spssxd_p.dll

You must provide the means for instances of the Processor class, from SPSS.BackendAPI.Controller.dll, to locate the IBM SPSS Statistics backend API library spssxd_p.dll. See the topic Chapter 1, “Getting Started,” on page 1 for more information.
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Software Group
ATTN: Licensing
200 W. Madison St.
Chicago, IL; 60606
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.
Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.
Index

Special characters

.NET
  class template 1
  DataCursor class 9
  DataCursorAppend class 14
  DataCursorWrite class 12
  EvaluateXPath method 19
  Processor class 1
  Submit method 5

A
  Append method 27

B
  BasePivotTable class
    Append method 27
    Insert method 27
    SetCategory method 28
    SetCell method 29
    SetCellsByColumn method 29
    SetCellsByRow method 29
    SimplePivotTable method 25

D
data
  appending cases 9, 14
  creating new variables 9, 12
  reading active dataset from .NET 9
dictionary
  reading dictionary information from .NET 7

I
  Insert method 27

O
  output
    reading output results from .NET 19
  OXML
    reading output XML from .NET 19

P
  pivot tables 25

R
  running command syntax from .NET 5

S
  SetCategory method 28
  SetCell method 29
  SetCellsByColumn method 29
  SetCellsByRow method 29
  SimplePivotTable method 25
  SPSS.BackendAPI.Controller.dll 1
  SPSS.BackendAPI.dll 1

T
  TextBlock class 33

V
  Visual Studio 1

X
  XML workspace 19
    writing contents to an XML file 19
  XPath expressions 19