# Collections for EasyLanguage

## Introduction

The ELCollections library supports two kinds of collections: Lists and Maps.

- A *List* is an ordered list of values. Values can be pushed onto the front or back of the list, and they can be popped off the front or back. They can also be inserted or removed at any position in the list. A value can be accessed by an index (like an array), or by looking at the front or back of the list. Lists can be "grown" on demand by adding or inserting values.

- A *Map* associates keys and values. A *key* is a piece of data for which you want to store an associated *value*. For example, suppose you want to store the last closing price for each symbol. You can put this information in a map, which can be represented conceptually like this:

  ```
  CSCO -> 19.74
  IBM -> 93.37
  MSFT -> 29.77
  ORCL -> 13.34
  ```

  In this case, the symbol name is the *key*, and the closing price is the *value*. Although this example associates strings with numbers, you can also associate numbers with numbers, numbers with strings, and strings with strings. In fact, you can even associate strings or numbers with other collections. This is a powerful capability which will be explained in more detail later.

All of the collection functions use the following naming convention:

```
ListX.Method(ID ...) or
MapXY.Method(ID ...)
```

where
*List* or *Map* indicates the collection type
*X* and *Y* indicate the type(s) of data in the collection:
N (number), S (string), or C (collection)
*Method* indicates the action to perform on the collection
*ID* is the identifier of the map or list

In the case of a *List*, *X* indicates the type of data in the list.

In the case of a *Map*, *X* indicates the key type, and *Y* indicates the value type.

### Examples

```
Count = ListN.Count(PriceList);
```
*Gets the number of values in a list of numbers (PriceList).*

```
Value1 = ListS.PushBack(SymbolList, GetSymbolName);
```
*Pushes the symbol name onto the back of a list of strings (SymbolList).*

```
Price = MapSN.Get(PriceMap, GetSymbolName);
```
*Gets the price associated with the current symbol from a map (PriceMap) that associates strings with numbers.*

### Valid Types

The following collection types are valid: ListN, ListS, ListC, MapNN, MapNS, MapNC, MapSN, MapSS, and MapSC.

MapCN and MapCS are not valid types because a collection is not a valid key type.

## The ELC Server

When you first use ELCollections in a study or strategy, you will see a program called "ELC Server" appear in the Windows task bar. This program manages memory for the ELCollections library, so it should always remain open while you are using ELCollections. When you close the last study or strategy that uses ELCollections, the ELC Server program will shut down automatically.

You don't normally need to worry about ELC Server at all; it just does its job in the background and disappears when it is no longer needed. However, if TradeStation exits abnormally for some reason, the ELC Server program may remain open. In this case, you should close it manually. To do this, just click the "ELC Server" button in the task bar and then click the close box in the upper-right corner of the window. You may also press Ctrl+C to close the server.

## Creating a Collection

There are two ways to create a collection: the New method and the Share method. Both of these methods return a numeric ID for the new collection. This ID must be stored in a variable and passed to any functions that operate on the collection.

The *New* method creates a collection that is visible only to the current study or strategy. It does not take any arguments.

The *Share* method creates a collection that can be shared by multiple studies or strategies. It takes a single argument, which is a unique name that identifies the collection. If no collection with that name exists, it is created; otherwise, the ID of the existing collection is returned.

### Examples

```
ListID = ListN.New; // creates a list of numbers
ListID = ListS.New; // creates a list of strings
MapID = MapSN.New; // creates a map of strings to numbers
MapID = MapNS.New; // creates a map of numbers to strings
```

```
// create a shared list of numbers called "Prices"
ListID = ListN.Share("Prices");

// create a shared map of numbers->numbers called "Distribution"
MapID = MapNN.Share("Distribution");
```

If a collection might be used on any bar of a study (as is usually the case), a good practice is to create the collection in the variable declaration, like this:

```
Vars:
   SymbolList(ListS.New),
   DistMap(MapNN.Share("Distribution"));
```

Remember that variables are initialized on the first bar, before any other code is executed. When the variable is initialized, the expression inside the parentheses is evaluated and assigned to the variable. Thus, the declaration of SymbolList above causes a new list to be created (by evaluating ListS.New), and the resulting List ID is assigned to the SymbolList variable. Since this happens on the first bar, the list is available for all the bars of the chart or RadarScreen, and its ID can be easily accessed via the SymbolList variable.

Likewise, the declaration for DistMap above causes the "Distribution" map to be created on the first bar (or to be shared if it already exists), and the resulting Map ID is assigned to the DistMap variable so that it can be easily accessed anywhere in the study or strategy.

## List Functions

Since EasyLanguage requires functions to return a value, all of the ELCollections functions must be assigned to a variable, even though many of them perform an operation rather than returning a meaningful value. In the following descriptions, if the return value is irrelevant, the function will be assigned to Value1. If the function returns a meaningful value, it will be assigned to a named variable such as MyValue.

Some of the following functions require an index. The index of the first value in a List is 1.

**`Value1 = ListX.Clear(ID);`**

Clears all the values from the list and frees the memory used by them.

**`Value1 = ListX.Resize(ID, NewSize);`**

Sets the size of the list to *NewSize*. If *NewSize* is less than the current size, the excess values at the end of the list will be deleted. If *NewSize* is greater than the current size, zeroes or empty strings will be added to the end of the list until it is the requested size.

Unlike the ListX.Clear function, the ListX.Resize function does not free all the memory used by the list. It will free any memory used by the values themselves (such as strings), but the space in the list will be reserved and reused if the list grows again. Thus, even though ListX.Resize(ID, 0) results in an empty list, it is not exactly equivalent to ListX.Clear(ID).

**`N = ListX.Count(ID);`**

Returns the number of values in the list.

**`MyValue = ListX.Get(ID, Index);`**

Returns the value in the list at the position specified by *Index*.

**`Value1 = ListX.Put(ID, Index, Value);`**

Puts *Value* into the position specified by *Index*, replacing the old value.

**`MyValue = ListX.Front(ID);`**

Returns the value at the front of the list, i.e. the first value.

**`MyValue = ListX.Back(ID);`**

Returns the value at the back of the list, i.e. the last value.

**`Value1 = ListX.SetBack(ID, Value);`**

Sets the back of the list to *Value*, replacing the old value.

**`Value1 = ListX.PushFront(ID, Value);`**

Adds *Value* to the front of the list. This automatically increases the size of the list by one.

**`Value1 = ListX.PushBack(ID, Value);`**

Adds *Value* to the back of the list. This automatically increases the size of the list by one.

**`Value1 = ListX.PopFront(ID);`**

Removes the value at the front of the list. Note that this function does not return the value. If you need to know the value, call Front before calling PopFront.

**`Value1 = ListX.PopBack(ID);`**

Removes the value at the back of the list. Note that this function does not return the value. If you need to know the value, call Back before calling PopBack.

**`Value1 = ListX.Insert(ID, Index, Value);`**

Inserts *Value* at the position indicated by *Index*. All existing values starting with *Index* are shifted up. This automatically increases the size of the list by one.

**`Value1 = ListX.Remove(ID, Index);`**

Removes *Value* at the position indicated by *Index*. All values above *Index* are shifted down.

**`Value1 = ListX.Rewind(ID);`**

Prepares to step through the values in the list using the ListX.Next function.

**`Continue = ListX.Next(ID, Value);`**

Gets the next value in the list. If there are any remaining values, the function sets *Value* to the next value and returns true. Otherwise, the function returns false. See below for an example of using this function to traverse a list.

**`Value1 = ListX.CopyFromList(ID, OtherList, Append);`**

Copies the contents of *OtherList* into the list. If *Append* is true, the contents of *OtherList* are appended to the end of the current list; otherwise, they replace the contents of the current list.

This function is not available for a ListC.

**`Value1 = ListX.CopyMapKeys(ID, MapID, Append);`**

Copies the keys from the specified map into the list (in sorted order). If *Append* is true, the keys are appended to the end of the current list; otherwise, they replace the contents of the current list. *MapID* must be the ID of a map with keys of the same type as the list (MapNC, MapNN, or MapNS for a ListN; MapSC, MapSN, or MapSS for a ListS).

This function is not available for a ListC.

**`Value1 = ListX.CopyMapValues(ID, MapID, Append);`**

Copies the values from the specified map into the list. If *Append* is true, the values are appended to the end of the current list; otherwise, they replace the contents of the current list. *MapID* must be the ID of a map with values of the same type as the list (MapNN or MapSN for a ListN; MapNS or MapSS for a ListS).

This function is not available for a ListC.

**`Value1 = ListX.SwapContents(ID, OtherList);`**

Swaps the contents of the list with *OtherList*. This operation is performed very quickly by the ELCollections DLL – it takes a small amount of time no longer how big the lists are. This is a fairly advanced function, but it is useful for solving certain types of problems efficiently.

This function is not available for a ListC.

**`Value1 = ListX.Sort(ID, Reverse);`**

Sorts the list. If Reverse is false, the list is sorted in ascending order. If Reverse is true, the list is sorted in descending order.

This function is not available for a ListC.

**`Value1 = ListX.SortByMap(ID, SortMap, Reverse);`**

Sorts the list using the associated values in *SortMap*. For each item in the list, this function looks up the associated value in *SortMap*, and the function sorts the list items by the associated values (rather than by the item values themselves). If *Reverse* is true, the list is sorted in descending order. See below for more information about this function.

This function is not available for a ListC.

```
Value1 = ListX.SortBy2Maps(ID, SortMap1, Reverse1,
    SortMap2, Reverse2);
```

Sorts the list using the associated values in *SortMap1* and *SortMap2*. This function works like ListX.SortByMap, but if two items in the list have the same associated values in *SortMap1*, the function uses the associated values in *SortMap2* to break the tie. *Reverse1* specifies the sort order for *SortMap1*, and *Reverse2* specifies the sort order for *SortMap2*.

This function is not available for a ListC.

```
Value1 = ListX.SortByNMaps(ID, SortMaps, ReverseFlags);
```

Sorts the list using the associated values in *SortMaps*. This function is a generalization of ListX.SortBy2Maps to any number of maps. *SortMaps* is a list of the maps to use. It will normally be a ListC, but it can also be a ListN that holds the IDs of the maps. (This is useful if these maps are already owned by another collection.) *ReverseFlags* is a ListN of numeric flags which specify the sort order for each corresponding map. A value of 0 in this list indicates ascending order, and a value of 1 indicates descending order. You may also pass a zero for this argument instead of a ListN; this tells the function to use ascending order for all the maps.

When two items in the list have the same associated values in the first sort map, the function tries to use the associated values in the second sort map to break the tie. If these associated values are also the same, the function tries to use the third sort map to break the tie, and so on.

This function is not available for a ListC.

```
Sorted = ListX.IsSorted(ID);
```

Returns true if the list is known to be sorted, which is the case when the list is empty, when it has just been sorted by ListX.Sort, and when all modifications to the list after either of these two conditions have kept the list sorted.

This function is not available for a ListC.

```
Index = ListX.InsertSorted(ID, Value);
```

Inserts Value into the correct location in a sorted list in order to keep it sorted. The function will raise a runtime error if the list is not already sorted. The function returns the index where Value was inserted.

This function is not available for a ListC.

```
Found = ListX.Lookup(ID, Value, IndexOut);
```

Performs a fast search for Value in a sorted list. The function will raise a runtime error if the list is not sorted. If Value is in the list, the function returns true and sets IndexOut to the

index of Value. If Value is not in the list, the function returns false and sets IndexOut to the index where Value should be inserted to keep the list in sorted order.

This function is not available for a ListC.

**`Index = ListX.Find(ID, Value);`**

Performs a linear search for the first occurrence of *Value* in a list. If *Value* is found, the function returns its index; otherwise, the function returns zero. Unlike ListX.Lookup, this function does not required the list to be sorted. (If you know that the list is sorted, you should use ListX.Lookup instead, since it is much faster.)

This function is not available for a ListC.

### Traversing a List with Rewind and Next

You can step through all the values in a list using the Rewind and Next functions:

```
Value1 = ListN.Rewind(ID);
while ListN.Next(ID, Number) begin
   // do something with Number
end;
```

You can also step through a list using a for loop:

```
for Index = 1 to ListN.Count(ID) begin
   Number = ListN.Get(ID, Index);
   // do something with Number
end;
```

The Rewind/Next technique is provided for consistency with maps. A map can be traversed with Rewind/Next but not with a for loop (since maps are accessed by arbitrary keys rather than by an index).

### Sorting by a Map

The ListX.SortByMap function is a very powerful function, but it is easier to understand with an example. Suppose you have a list called SymbolList:

```
SymbolList: CSCO, IBM, MSFT, ORCL
```

Suppose you also have a map called PriceMap:

```
CSCO -> 19.74
IBM -> 93.37
MSFT -> 29.77
ORCL -> 13.34
```

Now suppose you want to sort the SymbolList by the associated *prices* rather than by the symbol names. You can do this with SortByMap:

```
Value1 = ListS.SortByMap(SymbolList, PriceMap, false);
```

The SymbolList will now have the following order:

```
SymbolList: ORCL, CSCO, MSFT, IBM
```

You can sort by more than one map using the ListX.SortBy2Maps or ListX.SortByNMaps functions. If the associated values in the first map are the same, these functions use one or more additional maps to break the tie. See the descriptions of these functions above for details about how to call them.

## Map Functions

**`Value1 = MapXY.Clear(ID);`**

Clears all the key-value pairs from the map and frees the memory used by them.

**`N = MapXY.Count(ID);`**

Returns the number of key-value pairs in the map.

**`MyValue = MapXY.Get(ID, Key);`**

Returns the value associated with *Key*.

Remember that the key and value types depend on the map type. For example, for a MapSN, *Key* should be a string (S) and *MyValue* should be a number (N). For a MapNS, *Key* should be number (N) and *MyValue* should be a string (S).

There are two map types that can hold other collections: MapNC and MapSC. Since collections are represented by numeric IDs, these map types required a numeric variable for *MyValue*.

**`Value1 = MapXY.Put(ID, Key, Value);`**

Associates *Value* with *Key*. If *Key* is already in the map, the old associated value is replaced with *Value*. If *Key* is not in the map, the *Key*-*Value* pair is added to the map.

**`Value1 = MapXY.Remove(ID, Key);`**

Removes the key-value pair for *Key* from the map.

**`IsThere = MapXY.Exists(ID, Key);`**

Returns true if *Key* is in the map and false if it is not in the map.

**`Value1 = MapXY.Rewind(ID);`**

Prepares to step through the key-value pairs in the map using the MapXY.Next function.

**`Continue = MapXY.Next(ID, Key, Value);`**

Gets the next key-value pair in the map. If there are any remaining pairs, the function sets *Key* to the next key and *Value* to its associated value and returns true. Otherwise, the function returns false. See below for an example of using this function to traverse a map.

### Traversing a Map with Rewind and Next

You can step through all the key-value pairs in a map using the Rewind and Next functions:

```
Value1 = MapSN.Rewind(PriceMap);
while MapSN.Next(PriceMap, Sym, Price) begin
   // do something with Sym and Price
end;
```

## Cleaning Up Collections

Normally you don't have to worry about cleaning up collections yourself: the ELCollections utility will do it for you automatically when a study or strategy is unloaded. However, there may be times when you want to create a temporary collection, do some work with it, and then delete it to save memory. You can do this with the Release function:

```
Value1 = ListX.Release(ID) or
Value1 = MapXY.Release(ID)
```

If you release a local collection (created with New), it will be deleted immediately. If you release a shared collection (created with Share), it will be deleted only when it has been released by all studies and strategies that use it.

Here's a function called GetSymbolSet that creates a shared map of symbols by reading the symbols from a text file. (Reading from a file is discussed below.) In this case, the values associated with the symbols are not important – the map is just used to see whether the symbol is present using the MapXY.Exists function. This is much faster than searching through the list every time. Since the list is only needed once in order to build the map, it is released as soon as we are done with it:

```
Inputs:
  SetName(StringSimple),
  FileName(StringSimple);

Vars:
  SymbolSet(MapSN.Share(SetName)), // create map or share it if it exists
  TempList(0),
  Sym("");

// Only read the file the first time, i.e. when the map is empty
if MapSN.Count(SymbolSet) = 0 then begin
  TempList = ListS.New; // create a temporary list of strings
  Value1 = ListS.ReadFile(TempList, FileName); // read symbols from file
  Value1 = ListS.Rewind(TempList); // start from beginning of list
  while ListS.Next(TempList, Sym) begin // loop through the symbols
```

```
    Value1 = MapSN.Put(SymbolSet, Sym, 0); // put Sym->0 pair in map
  end;
  Value1 = ListS.Release(TempList); // delete the temporary list
end;
GetSymbolSet = SymbolSet; // return the map of Symbol->0 pairs
```

Note that this function just stores zero in the map for every symbol. In this case, we are not interested in the values which are paired with the symbols; the map simply provides a fast way to see if a symbol is present. Maps are designed to provide fast access to arbitrary keys, so looking up a symbol in a map is much faster than searching through a list of strings.

Once you have created a "symbol set" with the GetSymbolSet function, you can just use MapSN.Exists to determine whether a symbol is in the set:

```
SymbolSet = GetSymbolSet("EasyToBorrow", "C:\Temp\EB.txt");
EasyToBorrow = MapSN.Exists(SymbolSet, GetSymbolName);
```

The GetSymbolSet function demonstrates a good use for the Release function. The function only uses the list once (when the map is populated with data), so it makes sense to delete the list when the function is done with it. If you want to use a list or map on more than one bar, but you don't want to save the values across bars, a better solution is to call ListX.Clear or MapXY.Clear after processing the collection. This clears all the values, but the collection is still available to use on subsequent bars. This approach is much more efficient that creating the collection on every bar.

The general rule of thumb is this: if you only need a collection once, you can Release it when you are done with it (although you don't have to). If you want to re-use a collection without saving the values from bar to bar, call Clear when you are done with it.

## Collections of Collections

It is possible to create a collection that contains other collections. For example, you can create a Map of Lists, a Map of Maps, a List of Maps, or a List of Lists. This allows you to create sophisticated data structures to solve all kinds of problems.

Here is an example of a Map of Lists. The keys in the map are the date/time of each bar. The values in the lists are the open, high, low, and close. Thus, the map represents a series of price bars:

```
"2004/10/01 0635" -> (10.50, 10.65, 10.40, 10.60)
"2004/10/01 0640" -> (10.55, 10.60, 10.50, 10.58)
"2004/10/01 0645" -> (10.62, 10.67, 10.53, 10.65)
```

You could also represent this as a Map of Maps:

```
"2004/10/01 0635" -> (Open->10.50, High->10.65, Low->10.40, Close->10.60)
"2004/10/01 0640" -> (Open->10.55, High->10.60, Low->10.50, Close->10.58)
"2004/10/01 0645" -> (Open->10.62, High->10.67, Low->10.53, Close->10.65)
```

**Note:** Keep in mind that a map is a collection of key-value pairs. In each set of parentheses above, there are four key-value pairs, and together they comprise a map. Thus, the three lines shown above would constitute a single "master" map containing three sub-maps. Don't be confused because the pairs in the "master" map are shown in vertical sequence while the pairs in the three sub-maps are shown in horizontal sequence. It doesn't matter whether we group our pairs vertically or horizontally. "Pairs is pairs!"

To create a list of collections, use ListC.New or ListC.Share.

To create a map of collections, use MapSC.New, MapSC.Share, MapNC.New, or MapNC.Share.

Once you have created such a map or list, you can add collections to it using the New method.

**Note:** Do not use the Share method to create the collections which you add to a map or list. ELCollections will raise a runtime error if you try to add a shared collection to a collection.

Here is how you might create the map of lists shown above:

```
Vars:
   MapID(MapSC.New),
   ListID(0);

ListID = ListN.New;
Value1 = ListN.PushBack(ListID, 10.50);
Value1 = ListN.PushBack(ListID, 10.65);
Value1 = ListN.PushBack(ListID, 10.40);
Value1 = ListN.PushBack(ListID, 10.60);
Value1 = MapSC.Put(MapID, "2004/10/01 0635", ListID); // add list to map

ListID = ListN.New;
Value1 = ListN.PushBack(ListID, 10.55);
Value1 = ListN.PushBack(ListID, 10.60);
Value1 = ListN.PushBack(ListID, 10.50);
Value1 = ListN.PushBack(ListID, 10.58);
Value1 = MapSC.Put(MapID, "2004/10/01 0640", ListID); // add list to map

ListID = ListN.New;
Value1 = ListN.PushBack(ListID, 10.62);
Value1 = ListN.PushBack(ListID, 10.67);
Value1 = ListN.PushBack(ListID, 10.53);
Value1 = ListN.PushBack(ListID, 10.65);
Value1 = MapSC.Put(MapID, "2004/10/01 0645", ListID); // add list to map
```

Of course, it's usually more useful to build such a map algorithmically rather than manually. The following code will populate a shared map of lists with price bars from a chart. It assumes that we have a BarDateTime function that returns date/time strings like those above:

```
Vars:
   MapID(MapSC.Share(GetSymbolName + "_Bars")),
   ListID(0);
```

```
// This code runs on every bar, so it will add a new list
// of OHLC values to the map on every bar.
ListID = ListN.New;
Value1 = ListN.PushBack(ListID, Open);
Value1 = ListN.PushBack(ListID, High);
Value1 = ListN.PushBack(ListID, Low);
Value1 = ListN.PushBack(ListID, Close);
Value1 = MapSC.Put(MapID, BarDateTime, ListID); // add list to map
```

Finally, the following code will populate a shared map of maps with price bars:

```
Vars:
   MapID(MapSC.Share(GetSymbolName + "_BarMap")),
   PriceBar(0);

// This code runs on every bar, so it will add a new map
// of OHLC values to the map on every bar.
PriceBar = MapSN.New;
Value1 = MapSN.Put(PriceBar, "Open", Open);
Value1 = MapSN.Put(PriceBar, "High", High);
Value1 = MapSN.Put(PriceBar, "Low", Low);
Value1 = MapSN.Put(PriceBar, "Close", Close);
Value1 = MapSC.Put(MapID, BarDateTime, PriceBar); // add PriceBar to map
```

To access the high from a specific bar in the map of lists, you could use the following code:

```
Vars:
   MapID(MapSC.Share(GetSymbolName + "_Bars")),
   ListID(0);

ListID = MapSC.Get(MapID, "2004/10/01 0640");
Print("High = ", ListN.Get(ListID, 2)); // 1-4 = OHLC, per list layout
```

To access the high from a specific bar in the map of maps, you could use the following code:

```
Vars:
   MapID(MapSC.Share(GetSymbolName + "_BarMap")),
   PriceBar(0);

PriceBar = MapSC.Get(MapID, "2004/10/01 0640");
Print("High = ", MapSN.Get(PriceBar, "High"));
```

In summary, here are the guidelines for using collections of collections:

1. Create the collection with one of the following keywords: ListC.New, ListC.Share, MapSC.New, MapSC.Share, MapNC.New, or MapNC.Share.

2. To add a collection to the map or list, create the collection with one of the New functions, and then add it to the map or list in the usual way (e.g. Put or PushBack).

3. To access a collection within a map or list, use one of the usual methods (e.g. Get or Back) to retrieve the Collection ID into a variable. Then pass that ID to a collection function to perform the desired operation on the sub-collection.

## Type Checking

All collections have a "type" that is specified when you create the collection with New or Share. For example, if you create a collection with MapSN.New, then that collection has a type of "MapSN" – a map that associates strings with numbers. If you create a collection with MapSC.Share, then that collection has a type of "MapSC" – a map that associates strings with collections.

Once you have created a collection of a specific type, you must always use the correct type prefix when accessing that collection. If you don't, the ELCollections utility will issue a runtime error which will appear in the TradeStation Events Log.

For example, if you create a list using ListN.New, then you must use ListN.Get to get a value from that list. If you attempt to use ListS.Get instead, it will trigger a runtime error message.

If you have the ID of a collection, you can find out the type of the collection by calling ELC.CollectionType(ID). This returns a string that describes the collection type. For example:

```
Vars:
   FirstID(ListN.New),
   SecondID(MapSC.New),
   Type1(""),
   Type2("");

Type1 = ELC.CollectionType(FirstID);  // returns "ListN"
Type2 = ELC.CollectionType(SecondID); // returns "MapSC"
```

You don't normally need to use the ELC.CollectionType function, since you have usually created the collection in your code and therefore already know its type. However, this function can be useful if you have loaded a collection of collections from a file (see below), and you need to find out the types of the sub-collections.

## File Operations

There are a number of functions available which allow you to read a collection from a text file or write it to a text file.

These functions are very useful, but it should be stressed that they don't allow you to read and write *every* kind of collection. The variety of possible collection structures is virtually unlimited (since you can create collections of collections), so the ELCollections utility doesn't attempt to support them all. Instead, it allows you to read and write the most common structures, including certain forms of a map of lists or a list of lists.

Here is a description of the file functions, along with the supported file format for each function.

**Value1 = ListN.ReadFile(ID, FileName);**

Reads a list of numbers from a text file into the List specified by ID. The file should contain one number on each line, like this:

```
17.3
18.4
20.75
15
```

**Value1 = ListS.ReadFile(ID, FileName);**

Reads a list of strings from a text file into the List specified by ID. The file should contain one string on each line, like this:

```
CSCO
IBM
MSFT
ORCL
```

**Value1 = ListC.ReadFile(ID, FileName);**

Reads a comma delimited text file into the List specified by ID. The data is represented as a list of lists, where each column in the text file is imported as a sub-list. For example, suppose you have the following text file:

```
John,New York,31
Jane,Florida,25
Carlos,Oregon,19
Leticia,California,42
```

This would be read into the following list of lists (where the number on the left represents the index in the outer list):

```
1: John, Jane, Carlos, Leticia
2: New York, Florida, Oregon, California
3: 31, 25, 19, 42
```

The first and second sub-lists would be of type ListS, since the first and second columns in the text file contain strings. The third sub-list would be of type LisN, since the third column in the text file contains numbers. The ReadFile function automatically determines the type of list based on the first 20 values in the column.

**Value1 = MapNN.ReadFile(ID, FileName);**

Reads a list of key-value pairs from a text file into the MapNN specified by ID. Each line of the file should contain a numeric key and a numeric value, separated by a comma:

```
3.7,5.2
4.6,7.1
5.3,8.2
```

**Value1 = MapNS.ReadFile(ID, FileName);**

Reads a list of key-value pairs from a text file into the MapNS specified by ID. Each line of the file should contain a numeric key and a string value, separated by a comma:

```
30,Good
10,Bad
20,Indifferent
```

**Value1 = MapSN.ReadFile(ID, FileName);**

Reads a list of key-value pairs from a text file into the MapSN specified by ID. Each line of the file should contain a string key and a numeric value, separated by a comma:

```
Good,30
Bad,10
Indifferent,20
```

**Value1 = MapSS.ReadFile(ID, FileName);**

Reads a list of key-value pairs from a text file into the MapSS specified by ID. Each line of the file should contain a string key and a string value, separated by a comma:

```
CSCO,Long
ORCL,Short
MSFT,Flat
```

**Value1 = MapSC.ReadFile(ID, FileName);**

Reads a comma-delimited text file into the MapSC specified by ID. The first line of the text file should contain column headings. The data is represented as a map of lists, where each column heading in the text file is a key, and the rest of the column is imported as a list associated with that key. For example, suppose you have the following text file:

```
Name,State,Age
John,New York,31
Jane,Florida,25
Carlos,Oregon,19
Leticia,California,42
```

This would be read into the following map of lists:

```
Name -> (John, Jane, Carlos, Leticia)
State -> (New York, Florida, Oregon, California)
Age -> (31, 25, 19, 42)
```

The Name and State sub-lists would be of type ListS, since these columns in the text file contain strings. The Age sub-list would be of type ListN, since this column in the text file contains numbers. The ReadFile function automatically determines the type of list based on the first 20 values in the column.

**Value1 = ListN.WriteFile(ID, FileName);**

Writes the specified list of numbers to a text file. The file is written in the same format used by ListN.ReadFile.

**Value1 = ListS.WriteFile(ID, FileName);**

Writes the specified list of strings to a text file. The file is written in the same format used by ListS.ReadFile.

**Value1 = ListC.WriteFile(ID, FileName);**

Writes the specified list of lists to a comma-delimited text file. The file is written in the same format used by ListC.ReadFile. In order to write the list, it must meet the following conditions:

☑ The list contains only numeric lists (ListN) or string lists (ListS).

☑ All of the sub-lists contain the same number of values.

**Value1 = MapNN.WriteFile(ID, FileName);**

Writes the specified MapNN to a text file. The file is written in the same format used by MapNN.ReadFile.

**Value1 = MapNS.WriteFile(ID, FileName);**

Writes the specified MapNS to a text file. The file is written in the same format used by MapNS.ReadFile.

**Value1 = MapSN.WriteFile(ID, FileName);**

Writes the specified MapSN to a text file. The file is written in the same format used by MapSN.ReadFile.

**Value1 = MapSS.WriteFile(ID, FileName);**

Writes the specified MapSS to a text file. The file is written in the same format used by MapSS.ReadFile.

**Value1 = MapSC.WriteFile(ID, FileName);**

Writes the specified map of lists to a comma-delimited text file. The file is written in the same format used by MapSC.ReadFile. In order to write the map, it must meet the following conditions:

☑ The map contains only numeric lists (ListN) or string lists (ListS).

☑ All of the sub-lists contain the same number of values.

**Value1 = MapSC.SetColumnOrder(ID, ColList);**

You can call this function immediately before calling MapSC.WriteFile in order to set the order of the columns in the output file. (If you do not call this function first, the columns will be written in the alphabetical order of the keys.) *ColList* should be the ID of a ListS which contains the map keys in the desired column order.

It is not necessary to include all of the map keys in the column list. Any keys that are included will be written first in the specified order, and any remaining keys will be written as additional columns in alphabetical order.

If the column list contains any name that is not a valid key in the map, then the column list is ignored and the columns are written in the default alphabetical order.

Note that it is fine to release the column list (via ListS.Release) immediately after calling MapSC.SetColumnOrder.

**`Value1 = MapSC.BuildRowMap(ID, RowMapID, ColumnName);`**

This is not strictly a file function, but it was designed to work in conjunction with MapSC.ReadFile. Once a file has been read into a map of lists, it may be desirable to use one of the columns (lists) as an index into the rows. For example, once we have read the sample file above, we might want to use the "Name" column as an index, so that we can quickly look up a row by name.

The BuildRowMap function makes it easy to build a map of any column. Just pass the following arguments:

- ☑ The ID of the map which was read with MapSC.ReadFile.
- ☑ A RowMapID, which is the ID for a MapSN or MapNN that will map the column values to the row numbers. If the index column contains strings, you should pass a MapSN. If the index column contains numbers, you should pass a MapNN.
- ☑ The name of the column that you want to use as an index.

Once the row map has been built, you can look up the row number for any value in the index column. Then you can use this row number to retrieve values for the same row from other columns. If you read in a large file, this is much faster than a linear search through the column for the desired value.

Here's an example for the sample file above:

```
Vars:
   PeopleMap(MapSC.New),
   RowMap(MapSN.New),
   ColList(0),
   RowNum(0);

Value1 = MapSC.ReadFile(PeopleMap, "C:\Temp\People.txt");
Value1 = MapSC.BuildRowMap(PeopleMap, RowMap, "Name");

ColList = MapSC.Get(PeopleMap, "Age"); // get the Age column
RowNum = MapSN.Get(RowMap, "Carlos");  // get the row for Carlos

Print(ListN.Get(ColList, RowNum)); // print the age of Carlos
```

**Value1 = ListS.ReadHeadings(ID, FileName);**

Reads the first line of a tabular text file (in CSV format), parses the line into column headings, and populates the list with the headings.

This function is useful if you want to remember the column order of a tabular text file so that you can later write the data back with the same order. For example, the following code reads a text file into a MapSC and saves the column headings in a ListS. Later on, it uses the ListS to set the column order, and then it writes the MapSC to another text file. This ensures that the new text file will have the same column order as the original text file. (Remember that without the MapSC.SetColumnOrder call, the columns will be written in alphabetical order.)

```
Vars:
   InputFile("C:\Temp\Data1.csv"),
   OutputFile("C:\Temp\Data2.csv"),
   MyMap(MapSC.New),
   ColumnHeadings(ListS.New);

// Read the file and the column headings
Value1 = MapSC.ReadFile(MyMap, InputFile);
Value1 = ListS.ReadHeadings(ColumnHeadings, InputFile);

// Process MyMap in some way
...

// Write the map to a file using the same column order
Value1 = MapSC.SetColumnOrder(MyMap, ColumnHeadings);
Value1 = MapSC.WriteFile(MyMap, OutputFile);
```

**Value1 = ListS.FindFiles(ID, FilePattern);**

Populates a list with all the file names that match the specified file pattern. You can use this function to perform a directory search. The pattern should use the standard DOS wildcards * and ?. For example, the following code fills the DataFiles list with the names of all the files in the C:\Trading directory that match the Data*.csv pattern:

```
Vars:
   DataFiles(ListS.New);

Value1 = ListS.FindFiles(DataFiles, "C:\Trading\Data*.csv");
```

Note that the file names which are added to the list will not include the directory path – they only include the name and file extension (e.g. "Data10-22-04.csv").

## Specialized Functions

This section describes functions which are specialized for certain list types.

### ListN Functions

**Result = ListN.Summation(ID, FromIndex, ToIndex);**

Returns the sum of the values between *FromIndex* and *ToIndex* in the list.

**Result = ListN.Average(ID, FromIndex, ToIndex);**

Returns the average of the values between *FromIndex* and *ToIndex* in the list.

**Result = ListN.Variance(ID, FromIndex, ToIndex);**

Returns the population variance of the values between *FromIndex* and *ToIndex* in the list.

**Result = ListN.StdDev(ID, FromIndex, ToIndex);**

Returns the population standard deviation of the values between *FromIndex* and *ToIndex* in the list.

**Result = ListN.Maximum(ID, FromIndex, ToIndex);**

Returns the maximum of the values between *FromIndex* and *ToIndex* in the list.

**Result = ListN.Minimum(ID, FromIndex, ToIndex);**

Returns the minimum of the values between *FromIndex* and *ToIndex* in the list.

**Result = ListN.Percentile(ID, FromIndex, ToIndex, PctRank);**

Returns the specified percentile of the values between *FromIndex* and *ToIndex* in the list. *PctRank* should be a value between 0 and 1. For example, use 0.8 to calculate the 80$^{th}$ percentile.

**Result = ListN.Regression(ID, FromIndex, ToIndex, Slope, Intercept);**

Performs a linear regression on the values between *FromIndex* and *ToIndex* in the list, and returns the results in the *Slope* and *Intercept* arguments. The intercept is considered to fall at the beginning of the range (corresponding to *FromIndex*).

### ListS Functions

**Value1 = ListS.ParseString(ID, SourceStr, Delimiter);**

Parses a source string using the specified delimiter and populates a list with the resulting substrings. This allows you to use a string input to an indicator as a convenient way to specify a short list of strings. For example:

```
Inputs:
   Symbols("CSCO,IBM,MSFT,ORCL");

Vars:
   SymList(ListS.New);

Value1 = ListS.ParseString(SymList, Symbols, ",");

// Do something with SymList
...
```

### MapSC Functions

**`Value1 = MapSC.SortByCol(ID, KeyColName, Reverse);`**

Sorts the "rows" in a map by the values in one of the "columns". This function can be used to sort a map of lists, where each key in the map represents a column heading and the associated list represents the values in that column. All of the lists must contain the same number of items. (This is the same format used by MapSC.ReadFile and MapSC.WriteFile. See the description of MapSC.WriteFile above for an example of this format.)

This function performs the equivalent of sorting a data table in a spreadsheet by one of its columns. *KeyColName* is the name of the column that you want to sort by. If *Reverse* is true, the rows will be sorted in descending order; otherwise, they will be sorted in ascending order.

If the map contains any collections which are not lists, they will simply be ignored, and all of the lists will be sorted by the key column.

**`Value1 = MapSC.SortBy2Cols(ID, KeyColName1, Reverse1,`**
**`KeyColName2, Reverse2);`**

This function sorts the "rows" in a map by the values in two of its "columns". It is similar to MapSC.SortByCol, but if two values in the first key column are the same, the function uses the values in the second key column to break the tie.

**`Value1 = MapSC.SortByNCols(ID, KeyColNames, ReverseFlags);`**

This function is a generalization of MapSC.SortBy2Cols to any number of key columns. *KeyColNames* is a ListS containing the names of the key columns. *ReverseFlags* is a ListN of numeric flags which specify the sort order for each corresponding column. A value of 0 in this list indicates ascending order, and a value of 1 indicates descending order. You may also pass a zero for this argument instead of a ListN; this tells the function to use ascending order for all the columns.

When two rows have the same value in the first key column, the function tries to use the values in the second key column to break the tie. If these values are also the same, the function tries to use the third key column to break the tie, and so on.

## Utility Functions

The ELCollections library also provides some general utility functions. All of these functions are named with a prefix of "ELC.".

**`CharStr = ELC.AsciiToStr(AsciiCode);`**

Converts the specified ASCII code to a string. This is useful for creating characters that cannot normally be included in EasyLanguage strings. For example, you can use ELC.AsciiToStr(9) to create a tab character and ELC.AsciiToStr(34) to create a quote character.

**TypeStr = ELC.CollectionType(ID);**

Returns a string that describes the type of the collection represented by *ID*. For example, if the collection is a ListN, the function will return "ListN".

**Value1 = ELC.DirectoryCreate(DirectoryPath);**

Creates the specified directory. You must specify the full directory path. For example, the following code creates the directory C:\Temp\Data:

```
Value1 = ELC.DirectoryCreate("C:\Temp\Data");
```

**Value1 = ELC.DirectoryDelete(DirectoryPath);**

Deletes the specified directory. The directory must be empty, and you must specify the full directory path. For example, the following code deletes the directory C:\Temp\Data:

```
Value1 = ELC.DirectoryDelete("C:\Temp\Data");
```

**Value1 = ELC.PathExists(FilePath);**

Returns true if the specified file path exists. *FilePath* can be either a file or a directory. For example, the following code prints a message if a symbol file exists:

```
if ELC.PathExists("C:\Temp\Symbols.txt") then Print("Found symbol file");
```

The following code prints a message if the specified directory exists:

```
if ELC.PathExists("C:\Temp\Data") then Print("Data directory found");
```

**Value1 = ELC.RaiseError(AppName, ErrorMessage);**

Raises a runtime error. *AppName* should be a short string that describes your application. *ErrorMessage* is a message describing the problem. These strings will appear in the TradeStation Event Log.

It is strongly recommended that you use ELC.RaiseError rather than RaiseRuntimeError in any code that uses the ELCollections library. The RaiseRuntimeError can cause various problems, including loss of the original error message. This can make it very difficult to debug your code.