Introducing OOEL Concepts
by AndroidMarvin

Table of Contents

Introduction

For a fixed-function machine with limited controls, like a car or microwave, you don't need any grasp of how it works before you can master its controls. A programming language is just a complex interface to control a complex machine, giving an infinite range of possible functionality, and I firmly believe that although "somebody said you had to do it that way" will get you through simple tasks, to be creative or deal with complex tasks and errors, a grasp of the principles of how the language and computer operate - at a conceptual level but not a detailed level - is essential. It also reduces the memory load of having to remember lots of individual cases if you can see them as all following from the same conceptual mechanism.

The way I'll describe objects working, in relation to the programming language the end user types, isn't the only way they **could** work, but it is the way they do work in the majority of languages and Tradestation is following the norm. Anything I write about the way a Tradestation object works is the best understanding I have been able to build up, but realistically it is always possible that it is not true; I have no insider contacts. Please report any examples where a Tradestation object actually does something that conflicts with my theory, so I can correct the theory.

I will try and be as complete as I can, to make no assumptions about the prior knowledge any reader may have, so please be prepared to skip over material you already know - but if "what you know" conflicts with what I write, please consider reading some of the text you skipped in case that shows where the deviation comes from. There will probably be lots of cases where ideas A and B relate to each other, and if A is described first it will be necessary to go back and re-read A once B has been read.

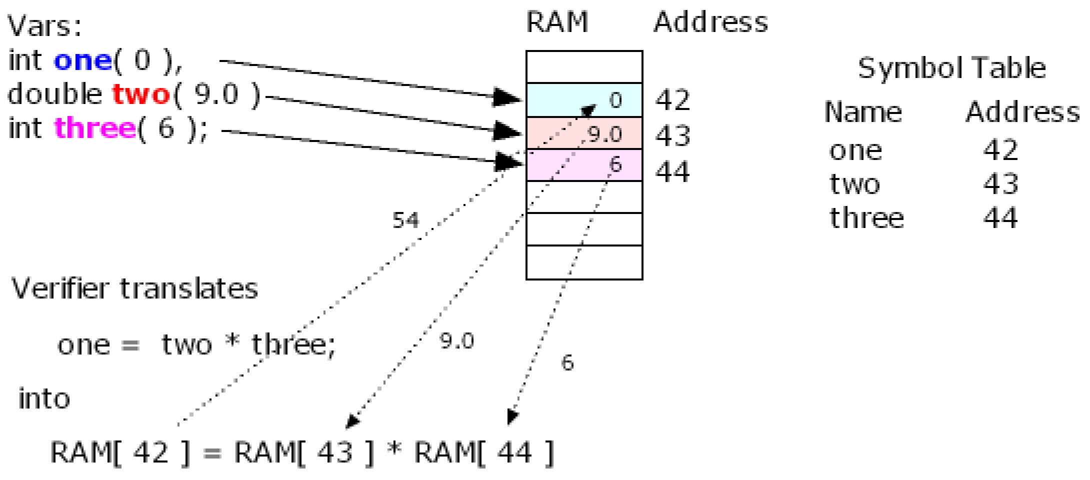
In examples, you'll find the word **aardvark** occurs frequently. I use this in place of the coy foobar (a mis-spelling of fubar) so beloved by C-derived books. Its a word that can't be mistaken for something you "have to type in order to get the code to work", and should be replaced by something representative of the specific problem being solved, like **getNextSetup**.

Blocks like this are "asides". They aren't part of the main thread, and can be skipped/disagreed with/laughed at as you wish.

Variables and pointers

A computer's RAM is like an enormous column of numbered boxes. When the processor wants to read from RAM, or save a value in RAM, it sends out a number representing the box it wishes to access; this is called an "address". (Each box holds one byte, whereas a variable usually needs 4 or 8 bytes to hold its value, and strings involve a different mechanism as they can be very large, so the processor will actually read or write "N bytes starting at address 42", where N depends on the type of variable; however, the discussion is simpler if we pretend that the whole of a variable's value is stored in the one box).

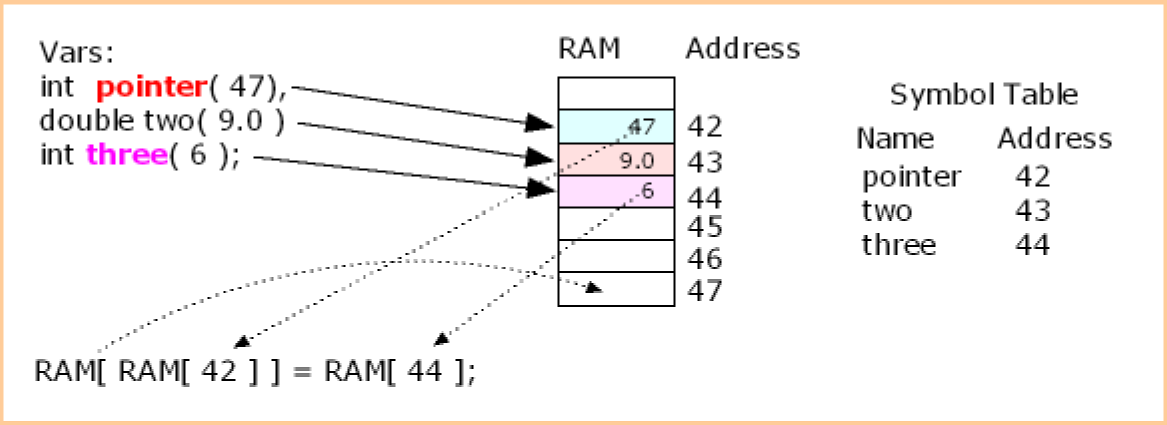
When you type a program into the Tradestation Development Environment, and click the Verify button, the Verifier has to apply the punctuation and spelling rules (the "**syntax** rules") of EL to figure out what the typed text means, then generate an equivalent set of instructions that the processor can obey in order to "execute" the program. One part of that process involves identifying the names that the programmer wants to use to represent the program's variables, and deciding the address where each variable will start in RAM. The verifier does this while analysing the **vars:** section of the program, and constructs a "symbol table" recording what address to use for accessing each variable.



This section can be skipped if you are happy to just accept that **a variable can contain a pointer to another place in RAM**.

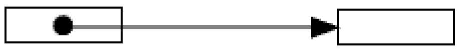
Notice that the **addresses** that identify places in RAM have the same format as **int** values; the only difference between the two is the way they are applied. The processor is perfectly capable of reading an **int** from one variable, then sending it out to RAM again as an **address**, and reading the contents of the box that address selects. Technically, this is called "indirection", but the most common term is "pointer".

EL doesn't allow you to explicitly write statements that can read an **int** and use it to select a place in RAM to be read from or written to, because such an ability would allow a programming error to manifest itself as a bug can be extraordinarily difficult to find, in which a "variable value" appears to change value without visible reason, either because the statement read from the wrong place or, worse, a statement elsewhere in the program wrote a new value into a place they shouldn't have. But the processor is perfectly capable of reading a number from RAM[42], then writing a value into RAM["the number we read from address 42"] - a value has been written into a "variable" that has no name.



This is a bit like a 'phone call forwarding/redirect/divert. You try to contact 'phone number 42, the exchange checks that 'phone and discovers that it has been set to redirect accesses to phone 47, so 47 is the phone you end up accessing.

In the rest of the document's diagrams, a pointer accessing another place will be shown as



Objects, method **out** parameters, and ....**Ref** inputs to functions all rely on this pointer mechanism, but restricted so that obscure bugs can't be created, as will be described shortly.

Functions

A function is a package containing a self-contained calculation, that will return "a result". The variables declared inside the function are completely inaccessible from outside the function, and the function can't directly access variables declared outside itself - except for the values provided for its declared inputs. This makes it quite difficult to use functions to implement multiple actions on a single body of data; each action needs to be a separate function, and to give them all access to the same data requires a potentially large number of parameters. Objects don't have this problem. It would be hard to provide a collection of data, with separate **Insert** and **Remove** operations provided by functions, in the way that a **Vector** object can.

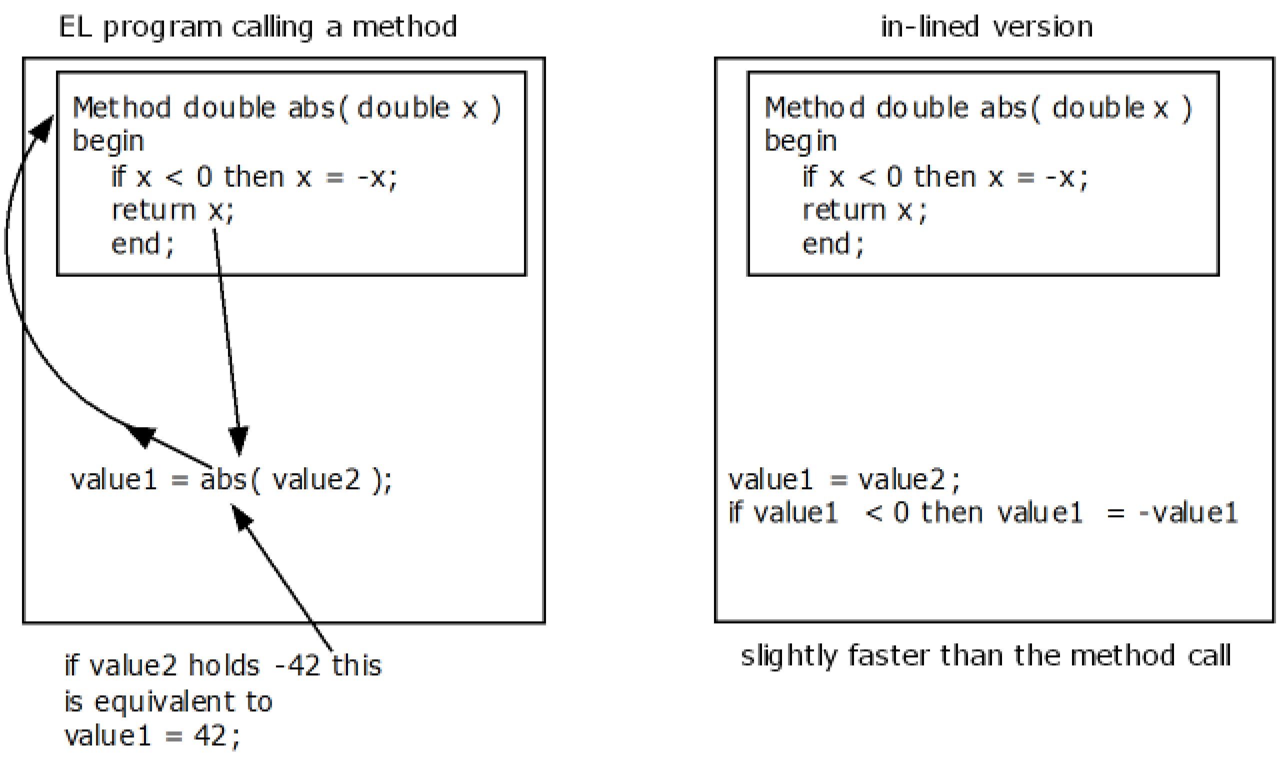
In the quest to make EL "easy" for its intended applications, EL functions have been made rather different to functions in any other language I know. I assume that when an input is declared as ....**Simple** it means that the main program gives the function a copy of a value to be worked with, and any changes made to that copy aren't visible outside the function, while an input declared as ....**Ref** is given a pointer (as described above) so that a change made in the function will be passed out to the caller's variable. Functions can have inputs declared as object types, can use OOEL internally, but can't return an object. Variables are given initial values when they are declared, but it seems that the variables are associated with the line that called the function, not with the function itself; the variables retain their value from one execution of a specific call to the next, they are not re-initialised on each call, and this seems to cause problems for some types of function and some types of call. The Wiki talks about "not calling functions inside a loop", for instance, and if a series function is called in the **then** part of an **if** statement, it may still be executed even when the **if** statement's condition is false.

Although functions provide a way to write some functionality once, and use it in many different programs, I don't understand their peculiarities well enough to feel safe trying to do anything but simple maths with them. If TS allowed users to define their own classes in separate files, or even included a mechanism for reading a file into the middle of another file during verification, using existing functionality in multiple programs would become easier.

Methods

A method implements an action that you can ask an object to perform; not the most sensible name, but its what the software industry uses. In OOEL, objects can have methods, but TS also uses the term **method** for a block of code in the EL "file" that can be invoked from more than one place in the program (presumably because they can't call it a "function" as that term already means something else).

TS tend to call it "an **in-line** method", or "in-line code", which it is when compared to a function (which is stored in a separate file), but in mainstream programming "in-line" is more likely to be something that is written out in full at the place where the action is required, to contrast that with putting the code for the action in a container (called a sub-routine, routine, or function depending on the language) so that control switches from the point where the action is needed to the place where the necessary code is stored, and returns once the code has completed its work. "In-lining" is a speed-up technique where the code of the routine is copied to the place where it is needed, to avoid the overhead of jumping from place to place in the program, and passing parameters; the opposite of the way TS uses "in-line".


EL program calling a method
Method double abs(double x)
begin
if x < 0 then x = -x;
return x;
end;
value1 = abs(value2);
if value2 holds -42 this is equivalent to value1 = 42;
in-lined version
Method double abs(double x)
begin
if x < 0 then x = -x;
return x;
end;
value1 = value2;
if value1 < 0 then value1 = -value1
slightly faster than the method call

The main benefit of methods to an EL programmer is that they allow an action to be applied at several different points in the program, without having to duplicate the code to carry out that action. A secondary benefit is that they allow a block of statements to be parceled up and given a descriptive name, allowing the code that applies the method to be short; understanding, and finding errors in, a block of code is easier if it can all be seen on the screen without scrolling, than if you have to scroll up and down to see the whole - I think

```
once initialise( );
if setupLongFound( details ) then
placeLongTrade( details )
else if setupShortFound( details ) then
placeShortTrade( details );
```

is easier to follow than if all the detail was presented in a block several screens long.

Like a function, an OOEL method can return "an answer", which is then effectively used in the calling statement in place of the text making the call. The type of returned value is specified by a reserved word that follows the reserved word **Method**. Thus

Method **int** aardvark would be the start of a method that returns an int value

This is similar to the selection of **Return Type** in a function's **Properties** dialog. Unlike a function, a method can return an object. Also unlike a function, the returned result can be ignored. So

```
index = aardvark( );
aardvark( );
```

are both equally acceptable to the Verifier (although they may not both be equally correct in your program, and you might prefer that the Verifier told you that you'd made a mistake by not using the returned result). Methods that have a return type are called like functions, by including their name in an expression within a statement; when the expression is evaluated, the returned value is used in place of the text of the method call. The call must include the ( ) after the method name, even if they do not enclose anything.

It is also possible to specify that a method does NOT "return an answer" that can be used in a statement; the word **void** means "there will be no returned value". Such methods are called by a statement that consists only of the method call, not as part of an expression. As above, the call must include the ( ) after the method name. So if method **aardvark** is declared

Method **void** aardvark( )

then

aardvark( );	would be a valid call statement for the method
index = aardvark( );	would **not** be a valid call statement

Because the method is not stored in a separate file, but is mixed into the text of an EL program, the executable statements that carry out the work of the method are enclosed between **begin** and **end** to separate them from the rest of the EL program.. Because (I assume) the main code of an EL program does not have an end marker, all method definitions must be completed before the start of "the main EL program". The order in which methods are defined doesn't matter; the first method you define can call the last that you define, without problems.

A method can define variables, using the **vars** keyword between the **method** statement and the **begin**. Unlike a function, variables are not, and cannot be, declared with an initial value in brackets. That means that each variable must be preceded by a data type keyword, as the

Verifier cannot infer the type of data the variable will hold from an initial value; I can only assume this is to save execution time as the assignment of an initial value would be carried out on each call of the method, while the initialisation of main program variables is only performed once? It looks as though local variables default to values

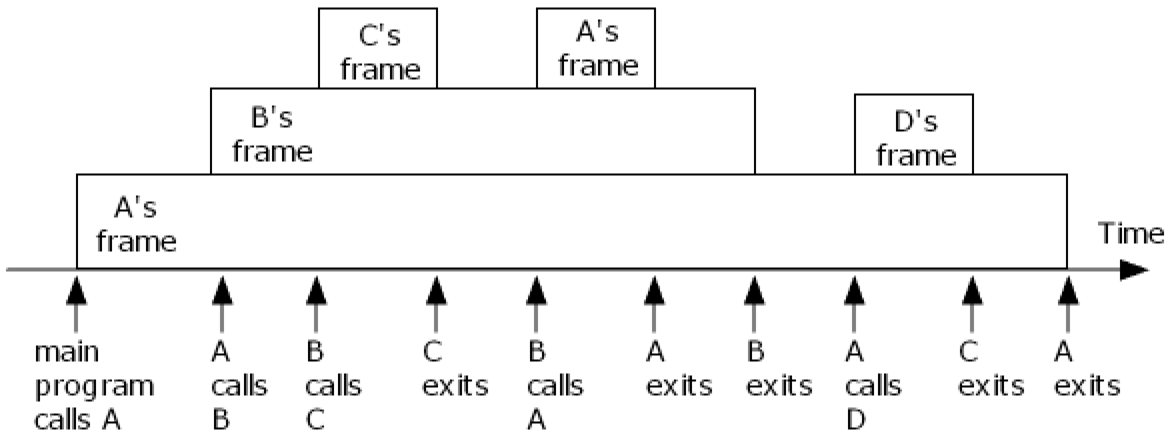
- 0 for ints
- 0.0 for doubles
- false for bools
- "" for strings
- **null** for objects

but I don't know if that is guaranteed.

Variables declared in a method are allocated RAM when the method is called, and that RAM is released when execution of the method completes; the variables don't retain values from call to call as those in functions do.

Following the practice of mainstream programming, EL methods have their variables allocated space on a data structure called a **call stack**. A stack is a last-in, first out store; newly added items are added to the top of the stack, and when an item is removed from the stack it is taken off the top of the stack. When a method is called, a stack frame is allocated, at the top of the stack, for the method to hold all its administrative details, including its local variables and parameters; when a **return** statement is executed, or execution reaches the final **end** of the method, the stack frame is removed from the stack. This technique is used, in general, to allow a method to call itself directly ("recursion"), or to be called more than once in a chain of calls; this is unlikely in EL - about the only place I can think of it being used is if a Vector contains an element that is a Vector (or a Dictionary contains a Dictionary) you might have a method that starts by processing the outer Vector or Dictionary, which calls itself if it finds that one of the items to be processed is itself a Vector or Dictionary. Perhaps processing XML documents would be another area where recursion might be useful.

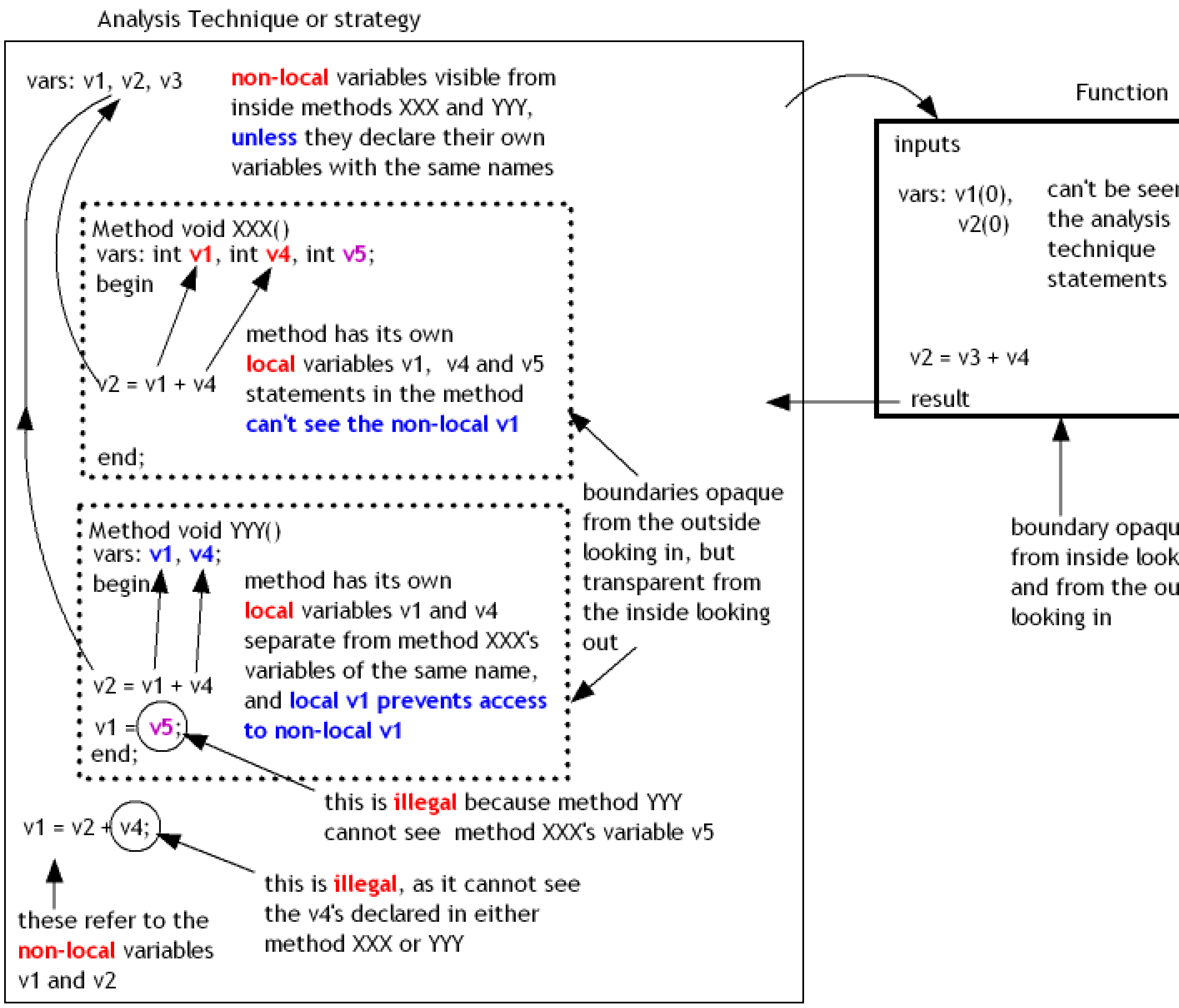
Notice that in the diagram there are times when two sets of the "variables declared by method A" exist in RAM.



If an exception occurs during the execution of a method, the Events Log will include, at the end of the Event details, a list of the names of the methods that were in execution at the time, by listing the frames on the call stack from top to bottom - ie starting with the method that caused the exception, then the method that called that, then the method that called that ... back to the main program.

The method is surrounded by a one-way glass wall - statements inside the method can see both variables declared inside the method and those declared in the main program; statements in the main program cannot see the names of variables declared inside the method. The range of lines from which statements can see a variable is referred to as the **scope** of that variable. The scope of a main program variable is from its declaration to the end of the file (you can't refer to the variable before it has been declared; **vars**: blocks can be placed between method definitions if you wish, but must occur before any reference to the variable sin the block); the scope of a local variable is from the **begin** to final **end** of the method definition within which the variable is declared - "**local** scope".

Variables declared in a method cannot be given initial values as part of the declaration


Analysis Technique or strategy
vars: v1, v2, v3
non-local variables visible from inside methods XXX and YYY, unless they declare their own variables with the same names
Method void XXX()
vars: int v1, int v4, int v5;
begin
v2 = v1 + v4
method has its own local variables v1, v4 and v5 statements in the method can't see the non-local v1
end;
boundaries opaque from the outside looking in, but transparent from the inside looking out
Method void YYY()
vars: v1, v4;
begin
v2 = v1 + v4
v1 = v5;
end;
method has its own local variables v1 and v4 separate from method XXX's variables of the same name, and local v1 prevents access to non-local v1
this is illegal because method YYY cannot see method XXX's variable v5
v1 = v2 + v4;
this is illegal, as it cannot see the v4's declared in either method XXX or YYY
these refer to the non-local variables v1 and v2
Function
inputs
vars: v1(0), v2(0)
can't be seen the analysis technique statements
v2 = v3 + v4
result
boundary opaqu from inside look and from the ou looking in

Mainstream programming would refer to the variables declared outside methods as **global**, and if the program made any of its variables visible outside the analysis technique they would be **public**. Unfortunately, within TS "global" has a history of use for "accessible outside the EL program", and references to several different mechanisms for achieving it, which is confusing. I'll try and always refer to **non-local**, or "**main program**" variables, but old habits are likely to trip me up.

For a method declared as returning **void**, control will return to the caller of the method when execution reaches the final **end;** of the method. It is also possible to return before reaching the **end;** by causing a **return;** statement to be executed. For a method declared as returning something other than **void**, then the **only** way to return from the method is to execute a **return** statement, which includes the value that is to be returned to the caller of the method; if the verifier detects that there is a way for execution to reach the final **end** without executing a **return**, it will report "not all control paths return a value".

For both types of method, the code can include more than one return statement, if desired - although some programmers don't like the use of multiple returns. It can be useful, though, for validation or to avoid extensive nested **if**s. Eg

```
Method void _am_sendTicket( string letter, OrderTicket ticket )
vars:
Order sentOrder;
begin
If chkPlaceOrders.Checked = False then return;
If OrderTicket.OrderPlacementEnabled = False then return;
If tradeAccount.RTDayTradingBuyingPower < 1000 then return;
If Time < startTime or time > endTime then return;
....
send the ticket to the servers, and do any administration
end;     // when there were no problems, return to the caller when execution reaches this line
```

It would, perhaps, be better in this case to have made the method return a value, say an **int**, and allow each return to return a different value so that the caller can tell whether the method did anything useful, or balked - and if so, for what reason.

Methods may have **parameters**. These are, effectively, **local variables**, but declared in a way that requires the caller of the method to **preset** them with initial values before execution of the method starts. The action is a bit like a calculator, where you must have a value in the display before pressing a function key like SQRT or SIN, then the "method" linked to the key takes the display value as a starting point and carries out its work, finally leaving its answer in the display in place of the starting value.

Parameters are declared in round brackets after the method name; the example above shows the **_am_sendTicket** method as having declared two parameters, one a **string** and one an **OrderTicket** object. The parameters have names, so that the statements inside the method can access the values of the parameters, but the outside world cannot see their names - the caller provides values of the right type, in the right order, between the ( ) following the method name in the call.

> Some people like to use words **parameter** and **argument**, one for the name that is visible inside the method, and one for the information passed from caller to the method. Personally, I've never found the distinction valuable enough to justify the effort of remembering which word refers to which concept, so I always use **parameter** from both viewpoints on what I consider the single passing mechanism. (And no doubt some people prefer to use argument for everything).

The most common style of passing parameters, shown above, results in the method having a **copy** of the parameter values - which it can then change internally without the caller being aware of the changes made. Suppose, for instance, that you wanted a method that would find the biggest High - Low range for a bar within a given number of bars. You might code that as

```
Method double maxRange( int length )     // method returns "an answer" of type double
vars:
int index,                               // index for scanning recent price history
double max,                              // biggest range found so far
int offset;                              // index of the biggest range found so far
begin
max = high[ 0 ] - low[ 0 ];              // assume the most recent bar is the biggest range
offset = 0;
For index = 0 to length - 1 begin        // then scan back looking for a bigger
If high[ index ] - low[ index ] > max    // if a new max has been found
then begin                               // save the new max's details
max = high[ index ] - low[ index ];
offset = index;
end;
end;                                     // specify "the answer value" that is to be returned to
return max;                              the caller
end;

If Barstatus( 1 ) = 2 then begin         // if this is the last tick of the bar
maxRange1 = maxRange( length1 );         // find the maximum range bar within one block of
maxRange2 = maxRange( length2 );         data
end;                                     //     "     "     "     second block of data
```

Suppose (for the sake of illustrating variations on a simple example, rather than because you might want to do it) that you wanted the **maxRange** method to work with whatever data happened to be available, rather than causing **MaxBarsBack** to increase. Then you might extend it to

```
Method double maxRange( int length )     // method returns "an answer" of type double
vars:
int index,                               // index for scanning recent price history
double max,                              // biggest range found so far
int offset;                              // index of the biggest range found so far
begin
if BarNumber = 1 then return high[ 0 ] - // illustrates leaving the method early, with an
low[ 0 ];                                answer
if BarNumber < length then length =      // changes the copy of the parameter, not the
BarNumber;                               caller's original
```

```
max = high[ 0 ] - low[ 0 ];                     // assume the most recent bar is the biggest range
offset = 0;
For index = 0 to length - 1 begin               // then scan back looking for a bigger
If high[ index ] - low[ index ] > max then      // if a new max has been found
begin                                           // save the new max's details
max = high[ index ] - low[ index ];
offset = index;
end;
end;                                            // specify "the answer value" that is to be returned
return max;                                     to the caller
end;
```

Now the first line causes the method to return immediately if its not appropriate to carry out the full calculation, while the second line prevents going back to far in the early bars of the chart; the method changes the value of its parameter, but when called by

```
value1 = maxRange( myChoice );
```

the caller's **myChoice** variable would continue passing in the same value on each of the earlier bars, although the method would actually be working with an effective parameter of 2, then 3, then 4, then 5 .... on bars before MaxBarsBack. (Its also possible to make the call "value1 = maxRange(42);" without worrying what would happen if the method tried to change the value of the literal 42).

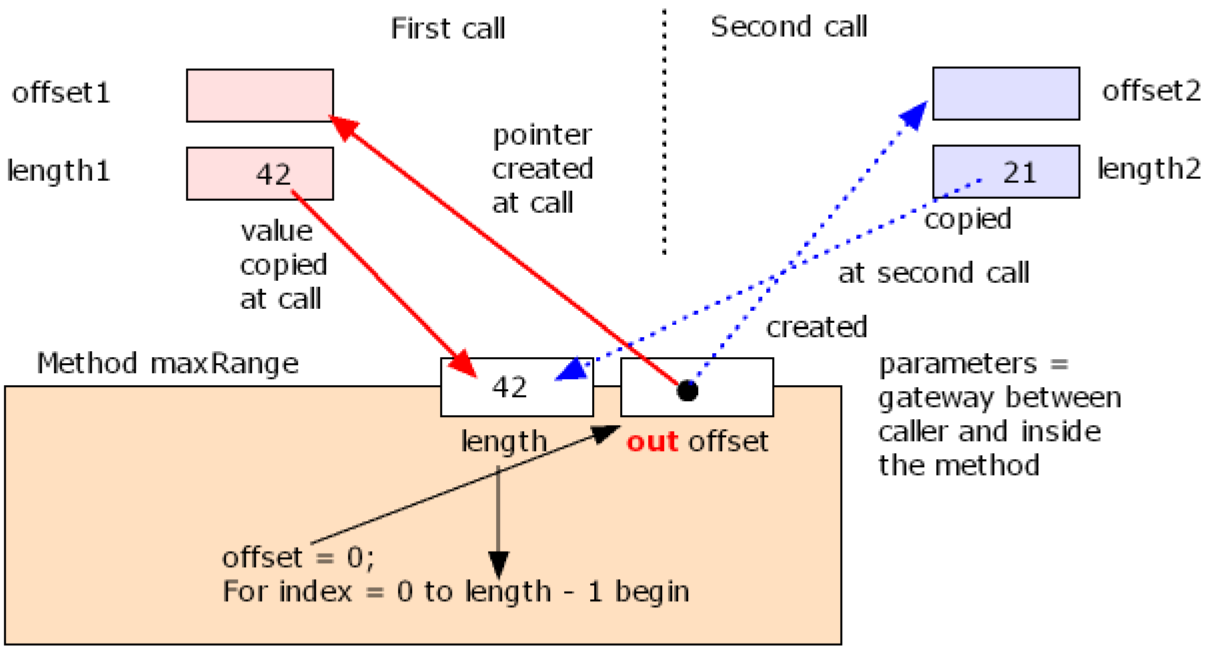
For a second variation, suppose the caller wanted to know not just the value of the highest recent range, but also how many bars back the maximum occurred. This could be achieved by using an out prefix on a second parameter.

```
Method double maxRange( int length, out         // method returns "an answer" of type double
int offset )
vars:                                           // index for scanning recent price history
int index,                                      // biggest range found so far
double max;
begin
max = high[ 0 ] - low[ 0 ];                     // assume the most recent bar is the biggest range
offset = 0;
For index = 0 to length - 1 begin               // then scan back looking for a bigger
If high[ index ] - low[ index ] > max then      // if a new max has been found
begin                                           // save the new max's details
max = high[ index ] - low[ index ];
offset = index;
end;
end;                                            // specify "the answer value" that is to be returned
```

```
return max;                                    to the caller
end;

If Barstatus( 1 ) = 2 then begin               // if this is the last tick of the bar
maxRange1 = maxRange( length1, offset1 );      // find the maximum range bar within one block of data
maxRange2 = maxRange( length2, offset2 );      //     "     "     "     second block of data
end;
```

Now, during the execution of **maxRange** from the first call, all references to **offset** inside the method will actually access the caller's variable **offset1**, and on return to the caller **offset1** will contain the distance from the current bar to the one with the maximum range; similarly while **maxRange** is being executed from the second call, all references to offset will actually refer to the caller's variable **offset2**.



Note that for objects as parameters, the passing mechanism works in the same way, but has a different effect because what is actually being passed is normally a **copy of a pointer** to the object **NOT** a copy of the object - so that the method **can** change values inside the object, even though there is **no out** prefix; if there is an **out** prefix, this gives the method permission to change the object variable - to make it point at a different object, or create an object inside the method and pass it out so the caller can use it. More detail later.

As an advanced point, it is technically possible to have **two methods with the same name** (unlike functions) - as long as the two have different numbers and/or type combinations for their parameters - the Verifier will work out which method to call based on the pattern of parameter types given in the call. This is called **overloading**. So, if a program included **both** the first **and** third versions given above for maxRange( ), then that program could legally include calls

```
maxRange1 = maxRange( firstLength );
maxRange2 = maxRange( secondLength, offset2 );
```

and there could be other versions with more parameters, or different type combinations such as Method void maxRange(double result, int length);

Classes and Objects

The original motivation for the "object-oriented" style of programming was to promote **software re-use** by allowing a block of related data and functionality to packaged into a form whereby, once tested and proved to be correct, it could be dropped into many different programs without the risk of it breaking (as so often happened when code was copied and pasted into different programs), ideally in a such a way that if the one copy of the original was modified, all the uses of that copy would be updated just by re-compiling, rather than requiring each program that uses it to be edited. The style has other characteristics, but they are relatively incidental to the re-use.

Tradestation has, so far, reserved this benefit for themselves (apart from the "tested and proved to be correct" part), customers cannot create their own classes to be re-used among the indicators and strategies they develop. Currently, the only real benefit users get from OOEL, apart from methods, is access to increased functionality without a huge increase in the number of reserved words and functions (and the corresponding difficulty of finding the right reserved word or function in a much longer alphabetical list than we have now, because all the new names needed are grouped inside containers so two containers can use the same new word rather than requiring two different reserved words to be invented), and more convenient ways of organising and grouping some data.

A **class** is a bit like a small factory, that knows how to manufacture one specific type of product. An **object** is a specific product after it has been manufactured. that a customer can take away and use. The class can be called on to manufacture as many objects as the user requires, that all have the same characteristics.

Objects don't have names. To access an object, you need to declare a variable (which can be referred to by name) or use a property of an object (that has already been declared, and has a name); you include the name in your EL statement, and the Verifier takes care of making that name access the object when the program runs.

A variable that will be used to give access to an object is a bit like a contact entry in a mobile phone; its a space that you can access by name, and inside that space there can be the extra information needed to establish contact with some sort of agent. Just creating a contact

"Broker" on your phone doesn't achieve much; to make it a useful contact, you have to store a 'phone number in the contact. Once the 'phone number is in place, you can ask the 'phone to go to the "Broker" contact and make contact with the actual person, so you can then ask them a question, or request an action. If you wish, you can change the 'phone number in the "Broker" contact, so that the next time you use the contact your 'phone will contact a different individual. In this analogy, the object is the person you call, and the 'phone number is "a pointer to an object". Before you can use a variable to access an object, the object must have been created in RAM, and a pointer to the object must have been copied into the variable.

The word "pointer" has had a lot of bad press over the decades. In the '70s and '80s, pointers were used to improve access speed; however, because a pointer was just an **int**, programmers chose to, or often had to, use arithmetic to manipulate the pointer - which was fine if it was done correctly, but errors in the arithmetic could easily lead to really obscure errors by accessing the wrong data values, resulting in a variable changing value despite never having been obviously referred to in a statement, or even writing data values on top of the program's executable instructions, so that the instructions in RAM were no longer the instructions that were generated by the verifier. Its common for programmers to talk about a variable holding "an object reference", but also "a statement references a variable", bringing slight touch of confusion (I think). The pointers used to access objects are much, much more tightly controlled than the pointers of the '70s, with none of the attendant risks, so I'm happy to use "pointer to an object", as a more direct description of the mechanism than the rather vague "reference" term.

If your program attempts to use objects created by a class, then the class will be loaded into RAM as part of loading the program. Only one copy of the class will exist, and you will access it by using the class' name. The class contains a template for creating an object in RAM, and a **Create** method that will allocate space for a new object in RAM and copy the template into the allocated space to create the new object. Sometimes the **Create** method has parameters that can be used to provide initial values for some aspects of the new object. Not all classes make their **Create** method visible to the EL program; the **Order** and **Position** classes don't, for instance, because it is only the TS platform that is allowed to create objects from these classes (in cooperation with the TS servers).

Classes and objects contain names for the properties and methods they contain. To access a method or property, you specify the container, then a dot, then the name of the method or property. Usually, the container will be an object; methods and properties made available as part of the class are referred to as static, because they exist at all times while the program is loaded (whereas objects are dynamically created on demand). In the Development Environment's Dictionary, if a method or property is described as **method public static** in the first line of the summary box in the bottom right pane, or if it starts property public and continues { **static read**; } then you access it by giving the class name; otherwise, access it by giving the object name. For instance, in the **elsystem.drawing.Color** class

Name	is summarised as	is accessed by
FromArgb	public method **static**	**Color**.FromArgb(....)
Black	public property .... { **static** read; }	**Color**.Black
B	public property .... { read; }	**myObject**.B

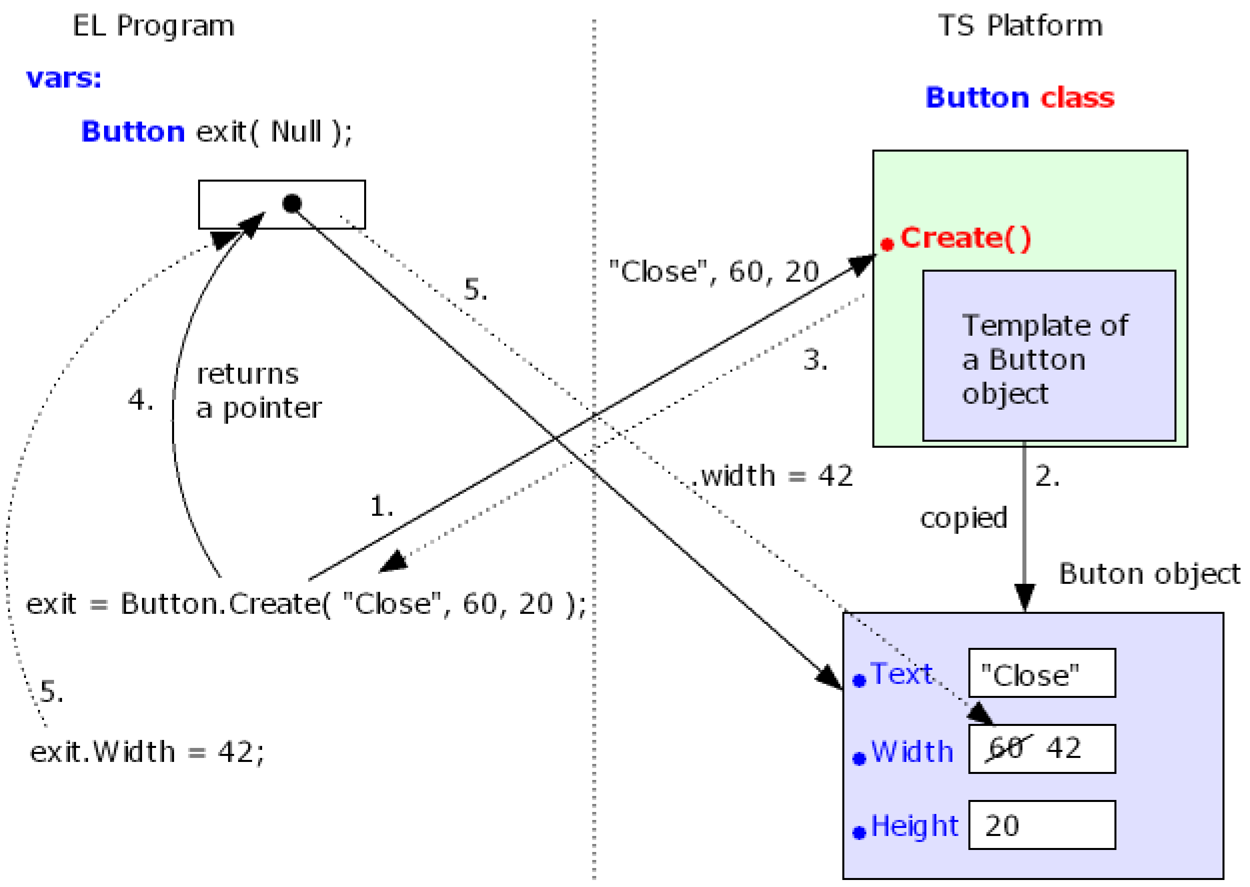
It would be nice if static methods and properties had different icons to non-static ones in the Dictionary, but they don't. It would be nice if something in the "Help" mentioned the meaning of the word "static", but I haven't found any such reference. It would be nice if lots of the Dictionary text was not just copies of fragments of the source code and pointless repetition without any serious thought of the reader's needs by "the Help team". It would be nice if .....

An object can be created by calling the class' **Create** method, or by using the **New** reserved word. Both create an object using the class' template, and return a pointer to the newly-created object. Often the object creation is part of an assignment statement, which saves the returned pointer in a named variable, which will later be used to access the object; its also legal to have the **Create** call or **New** in the brackets of a method call to create the object and pass it into the method for use. Personally, I mostly prefer to use the **Create** style rather than **New** because if the object creation can be influenced by parameters then **Create** gives a pop-up telling you what parameters you can provide, but I often use **New** for objects I use a lot that don't have parameters to their **Create** method, like **Vectors**.. **New** can be followed by the same parameters, but it's rather confusing syntax and there's no reminder; although **New** presumably actually calls **Create**. If there are no parameters then **New isn't** followed by **( )** while **Create** must always be followed by **( )** because it follows the syntax of method calls.

```
exit = Button.Create( "Close", 60, 20 );
exit = NewButton( "Close", 60, 20 );
```

both achieve the same result, but the .**Create** version seems more consistent with the rest of the EL syntax.

Objects created from the **Button** and **TextBox** classes can be added to a **Form** to allow the user to interact with the program. As with standard dialog boxes, a button can be clicked to trigger an action, the user can type text into the text box which the program can then read, but the program can also put text into the object for display. I'll use them as examples of simple objects.



In the diagram above, the variable **exit** initially holds **Null** to represent "doesn't point at anything". When execution reaches the statement that should create the **Button** object,

1. the EL program calls the static **Create** method of the **Button** class, passing parameters "Close", 60, 20.
2. the **Create** method allocates space for a new Button object in the platform's RAM, copies the template of a **Button** to the allocated RAM, and puts the parameter values into the new object.
3. the **Create** method returns a pointer to the newly-created object to the statement that called **Create**
4. the statement saves the pointer in the named variable **exit**
5. the main program can use the variable to change the value of the .**Width** property inside the object the variable points at; effectively,
    a. execution reads the pointer from variable **exit**, and follows it to an object
    b. the object is checked to see if it has a .**Width** property; if it hasn't, an exception is generated
    c. if the object has a .**Width** property, the new value is written into the property (subject to validation inside the object)

Initially, the variable **exit** is declared as "holding a pointer to a **Button** object", and will contain a code meaning "doesn't yet actually refer to a **Button** object"; that code has to be replaced by a pointer to an allocated **Button** object in RAM before the variable can be used. A pointer is actually just an **int**, but the Verifier places tight restrictions on how the pointer is included in statements, so that EL can never see the actual **int** value beneath the surface. This means that, actually, all "pointers to an object" are the same, regardless of the type of object they point at. When declaring a variable to point at an object, the Verifier asks you to specify the type of object it will point at (**Button**, here) simply so that it can check whether, when you put a dot after the **exit** name, that the name following the dot will be legal when the program runs. The type used to declare the variable is just your promise to the Verifier on what type of object the pointer will finally point at.

When a pointer to an object is declared in the main program, EL syntax requires that the declaration specify an initial value in **( )**; this is not required for a local method. The word **Null** was added to the EL language as a special code representing "a pointer that isn't actually pointing at anything", or just "no pointer". It can be used as the initial value of a declaration, as above, as it is compatible with all object types.

Once an object has been created, it stays in RAM, preserving its contents, until there are no longer any pointers to it. In .Net, when the system finds that there are no longer any pointers to an object it is marked as "garbage" and periodically a "garbage collection" process runs to reclaim the RAM used by such unused objects, so you can reduce a program's RAM footprint by statements like "myVector = **Null**" when the object is no longer needed. If an object is created in a method, and the pointer to it is stored in a local variable, unless that pointer is passed out to the main program, the object should disappear when control leaves the method, because local variables are destroyed at that point. Whether TS operates in the same way, I don't know - but I hope so. If a program fails to reclaim the RAM allocated to its objects, it constitutes a "memory leak".

Since objects have to be created before they can be used, if your program was to attempt to use the **exit** variable to access an object, before an object had actually been created and a pointer to the object placed in **exit**, you would get an exception "Object reference not set to an instance of an object" - ie "you've tried to follow a pointer, but the pointer is **Null**, so you can't go to what it points at". This is probably the most common error in OOEL, and the most frustrating because the message gives no clue to where the error is in the code. The Verifier can't help, because this error is only discoverable once the program is loaded on a chart and running.

You can guard against this error, by statements like

```
   if exit <> null then print( exit.Text );          // to prevent following a null pointer

or if exit = null then exit =                        // to detect that the object wasn't created,
   Button.Create( "Close", 60, 20 );                 and create it now
```

An object or static **method** is just like a method in the main program, except that it has access to the internal structure of the class or method. A **Property** appears to be a variable inside the object or class, that can be seen outside the object or class -but it isn't. Behind the syntax of "accessing a variable", the Verifier actually generates code to call a function to read a Property's value, or try to change a Property value. So when a programmer includes

myTextBox.Text = myTextBox.Text + " again";

in their EL, what gets executed is

myTextBox.**write**Text**(** myTextBox.**read**Text**( )** + " again" **)**; // a method call to read, and a method call to write, the .Text property

The benefit of this is that, when an attempt is made to change a property, the **write** method can validate the proposed change and reject it if it is inappropriate, it may update multiple variables inside the class or object to maintain their consistency, and the **read** method may in fact calculate the Property's value rather than simply reading a variable if that would be more efficient. The class or object also has the option of not providing a **write** method for a property, thus making it read-only (or, technically, not providing a **read** method to make it write-only, but that would be very unusual).

Going back to the TDE's Dictionary, if the Summary panel for a Property shows **{ write; read; }** in the second line, then the property value can be both read and changed. If the second line only shows **{ read; }** then you can read it but not change it - all the properties of **Order** and **Position** objects are read-only, for instance, as they are just displays of information coming in to your local PC from TS servers, and you wouldn't want your program to try and and change the Position's **OpenPL** value to make you more profitable, would you?

A similar behind-the-scenes method is an **indexer**. This selects one object within an object that contains many objects, but encloses its parameter in square brackets rather than the usual round brackets. So if your program uses an **OrdersProvider**, this seems to have two properties named **Order**, which seems a bit odd. In fact, **Order** is an indexer, and there are two overloaded methods with that name, one which accepts an **int** parameter and one that accepts a **string**.

The one that accepts an **int** parameter selects a specific order from within the **OrdersProvider** by its position in the sequence of orders being issued - myOP.Order[0] is the most-recently issued order, myOP.Order[1] next most-recently issued, etc. Depending on how the **OrdersProvider** is organised internally, this may achieve its result by simply accessing an element in an array using the index passed as a parameter as the index to the array. The myOP.Order that accepts a **string** parameter, however, definitely needs to be a method as it has to search its way through the list of **Order** objects recorded inside the **OrdersProvider**, checking each one to see if its **OrderID** matches the string parameter, and returning the **Order** object that matches as its result. The indexer approach has the benefit that it returns a specific type of object, whereas an item read out of a collection (q.v.) is just a general object, which is less convenient.

The last unexplained code, that I know of, that appears in the TDE Dictionary summary pane is **default**. This marks an indexer as being the indexer that will be used if the **[ ]** are placed immediately after the object name, without a "dot property name" prefix. For the **OrdersProvider**, the **Order** indexer that accepts an **int** parameter has a first line of "property public **default** tsdata.trading.Order **Order**[**int** index]" in the summary pane, meaning that

```
        myOP[ 42 ]
and     myOP.Order[ 42 ]
```

have exactly the same effect. Similarly, the "int parameter" **Position** indexer of the **PositionsProvider** is the default, so **myPP[0]** is the same as **myPP.Position[0]**, and for a **Vector** the **Items** indexer is the default indexer (although in this case its also the only indexer), so **myVec[42]** has the same effect as **myVec.Items[42]**. (Not all objects that contain objects have a default indexer, but many do.)

"Objects can be both passed to methods as parameters", and returned from methods as results; what is actually passed or returned is a pointer to the object. Forms can be quite repetitive to create, so I've found it helpful to have some standard "control creation" methods to avoid typing almost the same thing over and over. For instance,

```
Method Button newButton( string text, int width, int height, int x, int y )   // create a basic Button control
vars: Button b;
begin
b = Button.Create( text, width, height );
b.Location( x, y );
Return b;
end;
```

Would allow me to create a form with three buttons by

```
          //            Text      Width  Height    X    Y
btnYes    = newButton(  "Yes",    60,    20,      10,  10 );
btnNo     = newButton(  "No",     60,    20,      90,  10 );
btnCancel =             "Cancel"  60,    20,     170,  10 );
```

```
        newButton(
frmMain.AddControl( btnYes );
frmMain.AddControl( btnNo );
frmMain.AddControl( btnCancel );
```

Gathering all the numbers together in this way makes it easier to adjust the form layout, I find.

I also have an overload of **newButton**, that accepts **color** parameters when I want to use them :

```
Method Button newButton( string text, int width, int height, int x,     // create a Button control
int y, Color background, Color textColor  )                             with colors
vars: Button b;
begin
b = Button.Create( text, width, height );
b.Location( x, y );
b.BackColor = background;
b.ForeColor = textColor;
Return b;
end;
```

so now the **btnCancel** creation above could be

```
btnCancel = newButton( "Cancel", 60, 20, 170, 10, Color.Red, Color.White );
```

while the other statements remain the same.

For an **int** variable, the value assigned to the variable is held completely inside the variable. You can assign the same value to another **int** variable, so it holds a second copy of that variable, but the two copies remain completely independent. After

```
firstInt = 42;
secondInt = firstInt;
firstInt = firstInt * 2;
print( secondInt:6:0 );
```

you would, of course, expect to see 42 printed in the PrintLog. Although you can do simple work in OOEL while believing that the whole of an object is stored inside the variable you assign it to, that belief is false and sooner or later will trap you. The object equivalent version of that code (showing only the relevant properties) would be

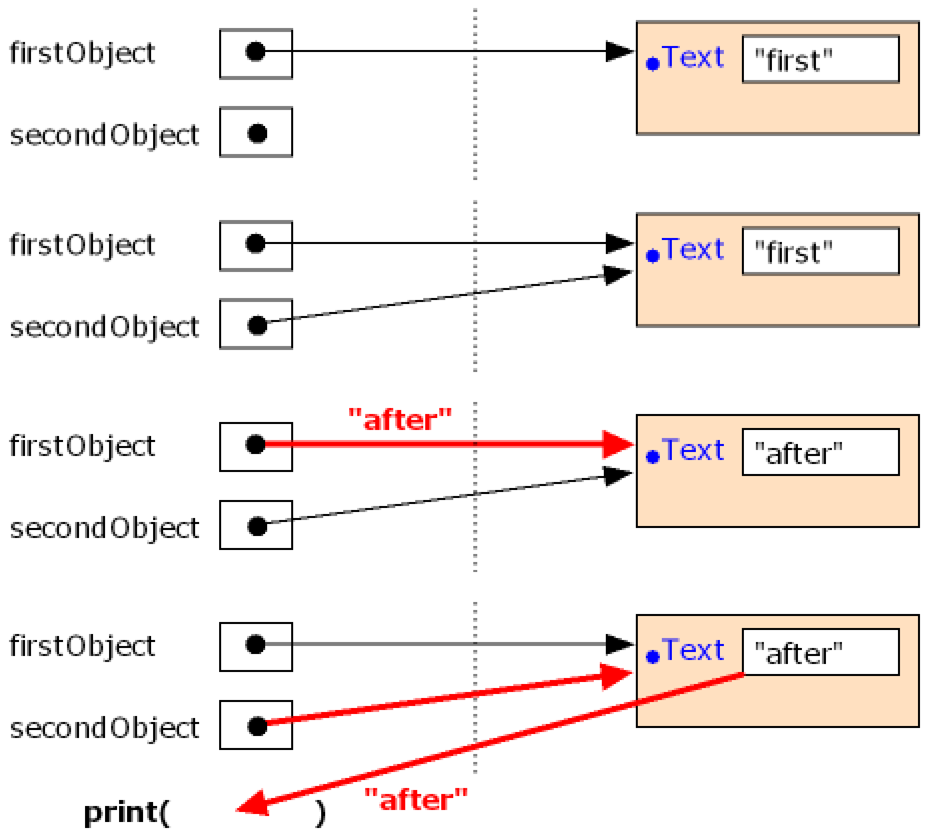
After executing   RAM contains

firstObject = TextBox.Create("first ", 60, 20);

secondObject = firstObject;

firstObject.Text = "after"

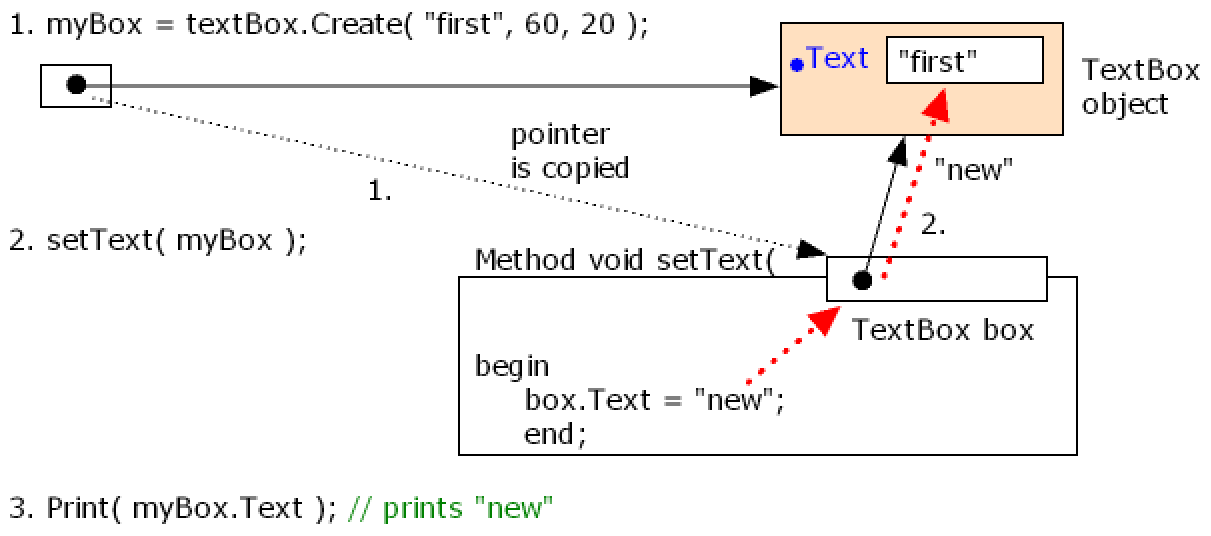
print(secondObject.Text);



1. initially, a **TextBox** object is created, **firstObject** is set to point at it, and the string "first" is copied into the object's **Text** property
2. the pointer from inside **firstObject** is copied into **secondObject**; they now point at the same object
3. the assignment follows the pointer from variable **firstObject** to the actual object, and copies string "after" into the destination's **Text** property
4. the print( ) statement follows the pointer from **secondObject**, retrieves the string from the destination's **Text** property, and prints it

What appears in the PrintLog is "after", because the two object variables point at the same object - not "first" as it would if "**secondObject** = **firstObject**;" copied an object stored inside **firstObject** into **secondObject**. Assigning one object variable to another **copies the pointer** from one to the other, not the object. Some, but not many, objects do have a .**Clone** method that can be called to create an independent copy of the object, if that is what you need.

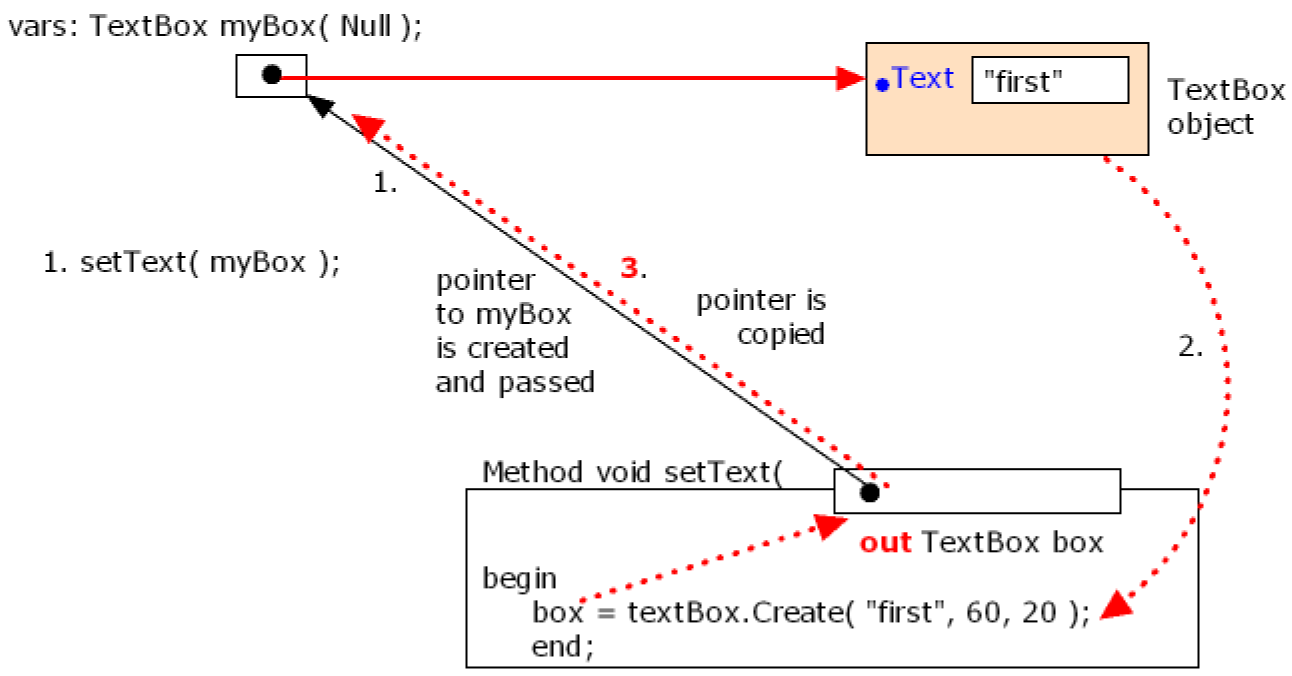
When "passing an object" to a method as a parameter, the same trap awaits. Simple parameters are normally copied into the method (and any changes to it are not made available to the caller) but an **out** prefix allows the method to reach out to the caller's variable and change it, so the caller can see a change made by the method - **BUT** for object parameters these rules apply to **the pointer to the object**, not to the object itself. To illustrate (with two rather useless examples) :



The method **setText** has a parameter that is a **TextBox**.

1. The main program creates a **TextBox** object and saves the pointer to the object in the named variable **myBox**.
2. It then calls the **setText( )** method, passing **myBox** as the parameter; this copies the pointer from **myBox** into the parameter local variable **box** that the method can use. The method then follows the pointer in its parameter **box** to the actual object and deposits "new" into the object's **Text** property.
3. The print( ) statement in the main program follows the pointer from **myBox** to the object, reads the **Text** property, and prints "new" - the object's contents have been changed even though the parameter did not have the **out** prefix.

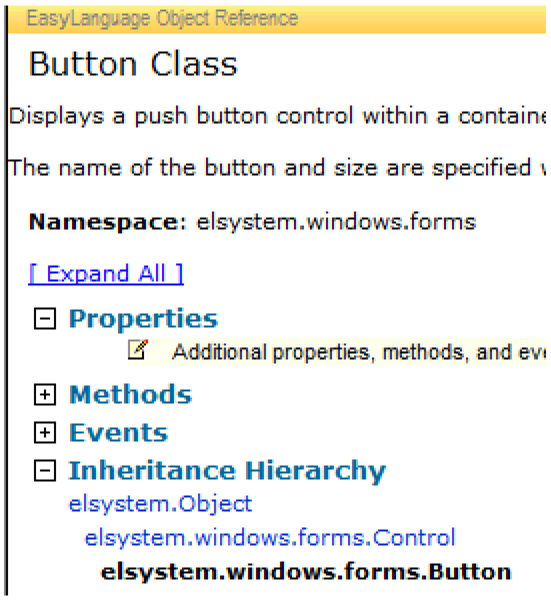
Giving an object parameter an **out** prefix gives the method permission to pass out an pointer to an object, not "permission to write to the object it was given".



1. the EL program calls a method that has an **out TextBox** parameter. A pointer to the caller's variable is created and passed to the method as a parameter .
2. the method creates a **TextBox** object, and receives a pointer to that object. If needed, it could use that pointer to access the properties of the new object before returning to the main program
3. when the method sets an object pointer into the parameter **box** that pointer is copied out into the variable the caller used as part of the call; here, the method created an object and gave the new object to the main program, but a different method could receive a pointer to an object from the caller, then shift that pointer so that it points at a different object.

Inheritance

In pursuit of the goal of writing code once and applying it whenever needed, Classes are not completely self-contained - a class can build on simpler classes that have already been created. This can be done by including objects created from one class in another class ("aggregation"), or by allowing a class to add features to another class to create an enhanced version ("inheritance"). In the general "Help" system this is shown in a separate section


EasyLanguage Object Reference

Button Class

Displays a push button control within a containe

The name of the button and size are specified w

Namespace: elsystem.windows.forms

[Expand All]

⊟ **Properties**

Additional properties, methods, and ev

⊞ **Methods**

⊞ **Events**

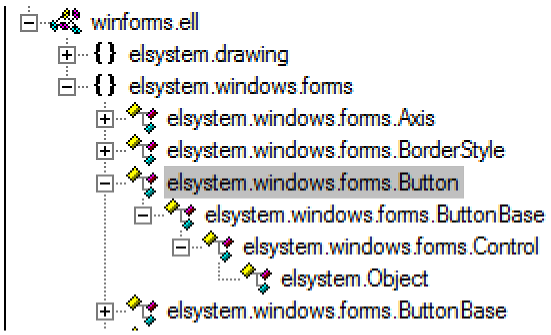
⊟ **Inheritance Hierarchy**

elsystem.Object

elsystem.windows.forms.Control

elsystem.windows.forms.Button

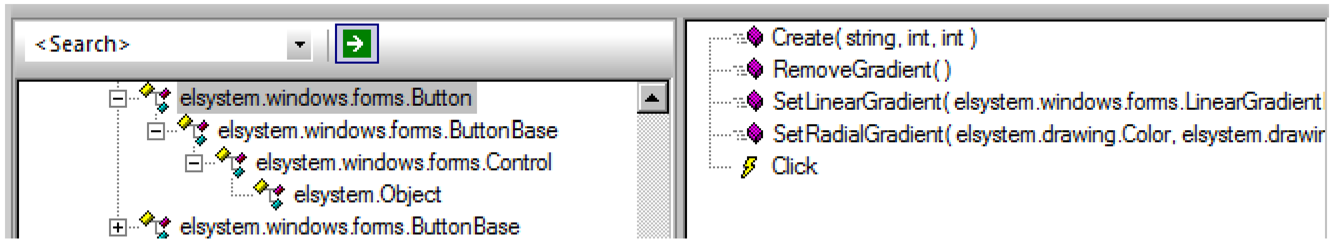
meaning that the **Button** class inherits some functionality from the **Control** class. and the **Control** class inherits some functionality from the **Object** class (which is the common ancestor of all classes, that establishes the basic concept of objectness and what it means). In the TDE Dictionary, the same thing (more or less) is shown in the tree pane on the left


```
⊟ winforms.ell
  ⊞ {} elsystem.drawing
  ⊟ {} elsystem.windows.forms
    ⊞ elsystem.windows.forms.Axis
    ⊞ elsystem.windows.forms.BorderStyle
    ⊟ elsystem.windows.forms.Button
      ⊟ elsystem.windows.forms.ButtonBase
        ⊟ elsystem.windows.forms.Control
            elsystem.Object
    ⊞ elsystem.windows.forms.ButtonBase
```

Which shows that the **Button** class inherits the functionality of the **ButtonBase** class and adds to it, **ButtonBase** inherited functionality from the **Control** class and added to it, and **Control** inherits functionality from the **Object** class and adds to it.

Inheritance is largely for the benefit of people creating classes, which EL programmers cannot do. It does tell the EL programmer that if a **Button** class and a **Label** class both have a **Text** property, it probably behaves the same way in both because both classes inherit from the **Control** class - but not necessarily; a class that inherits a **Text** property can, if it wishes, keep that **Text** property secret and create a property of its own which it then calls **Text**, which has different behaviour to the inherited **Text**. An example of this would be the **ToString**( ) method, which as defined in **Object** reports the **type** of an object, but some classes provide their own implementation of the **ToString**( ) they inherit, so that their version reports a **value** for the object. For instance a **DateTime** object's .**ToString**( ) method doesn't return "elsystem.DateTime", it returns something like "09/02/2013 18:21:17".

The Dictionary shows that an object created from the **Button** class has a **Create** method, (and, in Update 17 three other methods), and an event (to be discussed later).

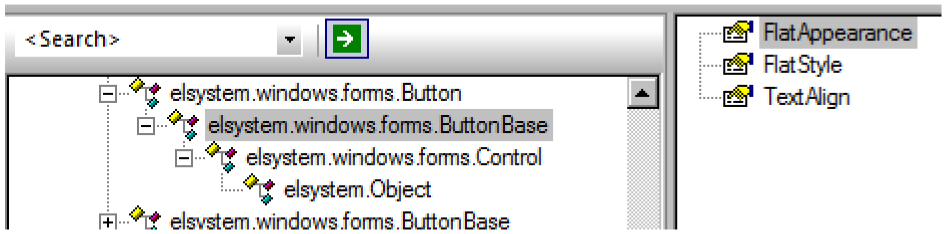


```
<Search>
elsystem.windows.forms.Button
  elsystem.windows.forms.ButtonBase
    elsystem.windows.forms.Control
      elsystem.Object
elsystem.windows.forms.ButtonBase
```

```
Create( string, int, int )
RemoveGradient( )
SetLinearGradient( elsystem.windows.forms.LinearGradient
SetRadialGradient( elsystem.drawing.Color, elsystem.drawir
Click
```

The **Button** class has a **Create** method, so you can create **Button** objects. The three "gradient" methods wouldn't have been my personal choice of a good way for TS to spend their development time, and putting them specifically in **Button** rather than something more high-level like **Control** seems a bit restrictive; we'll see if they move in the future.

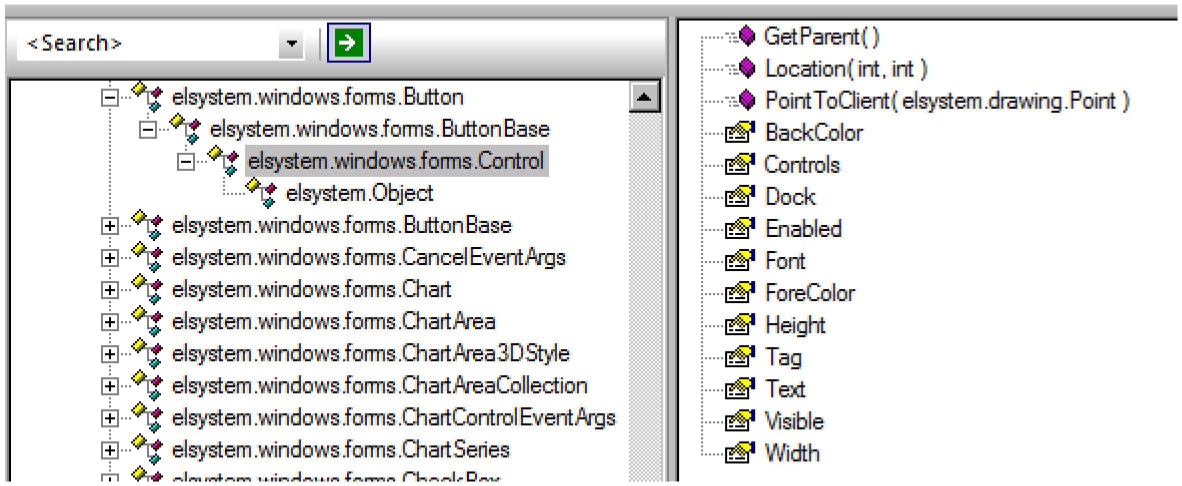
The **ButtonBase** class provides three properties, which every **Button** object will inherit



```
<Search>
elsystem.windows.forms.Button
  elsystem.windows.forms.ButtonBase
    elsystem.windows.forms.Control
      elsystem.Object
elsystem.windows.forms.ButtonBase
```

```
FlatAppearance
FlatStyle
TextAlign
```

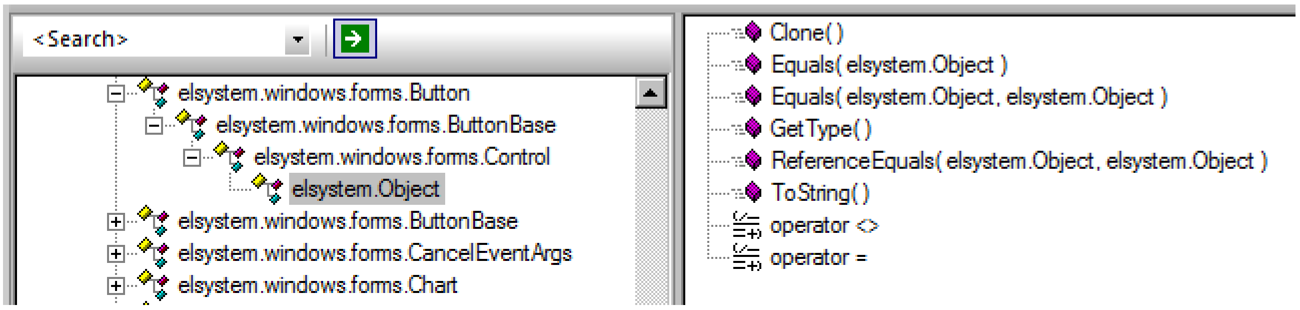
The **Control** class is the main source of **Button**'s functionality :



```
<Search>
elsystem.windows.forms.Button
  elsystem.windows.forms.ButtonBase
    elsystem.windows.forms.Control
      elsystem.Object
elsystem.windows.forms.ButtonBase
elsystem.windows.forms.CancelEventArgs
elsystem.windows.forms.Chart
elsystem.windows.forms.ChartArea
elsystem.windows.forms.ChartArea3DStyle
elsystem.windows.forms.ChartAreaCollection
elsystem.windows.forms.ChartControlEventArgs
elsystem.windows.forms.ChartSeries
elsystem.windows.forms.CheckBox
```

```
GetParent( )
Location( int, int )
PointToClient( elsystem.drawing.Point )
BackColor
Controls
Dock
Enabled
Font
ForeColor
Height
Tag
Text
Visible
Width
```

which are available to all the other classes that inherit from **Control**, and so work the same way (probably) in those. And the **Object** class offers



```
<Search>
elsystem.windows.forms.Button
  elsystem.windows.forms.ButtonBase
    elsystem.windows.forms.Control
      elsystem.Object
elsystem.windows.forms.ButtonBase
elsystem.windows.forms.CancelEventArgs
elsystem.windows.forms.Chart
```

```
Clone( )
Equals( elsystem.Object )
Equals( elsystem.Object, elsystem.Object )
GetType( )
ReferenceEquals( elsystem.Object, elsystem.Object )
ToString( )
operator <>
operator =
```

These are the basic functions that all objects have, such as a way of getting some sort of string representation, and the ability to compare an object to another. Notice that only the **Button** class has the **Create** method to allow the EL programmer to work with objects of the class; the inheritance chain shows how the button class was built up, from the TS program developer's viewpoint. A **Button** object has all of these properties and methods available, as shown in the autocomplete list, but to find such details as "Help" may offer on their functions, you have to explore along the inheritance tree.

AsType and IsType

Strictly, since the **Text** property of both a **Button** and a **Label** is inherited from class **Control**, it would be possible to code a method

```
Method void aardvark( Control param )
begin
print( param.Text );
```

then to call it by

```
aardvark( myButton );
aardvark( myLabel );
```

I haven't found a use for this for the OOEL programmer yet, given the classes available, but maybe one day? However, there are places (parameters to event handlers, and the contents of collections, for instance) in OOEL where you must deal with something that was declared as type **Object**. Since this is the common ancestor in the family tree of OOEL classes, this means that your code may have to deal with, potentially, absolutely any type of object. **AsType** and **IsType** help to deal with this.

IsType allows you to test, while the program is running, the type of object a pointer actually refers to before you try and access it, so you only try to access a property of the object a variable points at if the object actually has such a property, to avoid generating an exception. For instance

```
if myObject istype Button then myObject.Text = .....
```

This is important when using collections, which can hold any type of object, either to guard against a programming error having put the wrong type of object into the collection, or to process the collection contents correctly if you have designed a program that deliberately puts different types of object into one collection.

AsType doesn't, as far as I know, actually do anything while the program is running, but is important; like the object type you specify when declaring a variable, **AsType** tells the Verifier what type of object you expect a pointer to point at, so that the Verifier can check that any properties or methods of that object which you refer to in your EL statements, are consistent with the type of object that the pointer will connect the statement to. It does **NOT**, as far as I know, **convert** an object from one type to another (which would involve physical changes to the objects structure), So you might write

```
if myObject istype Dictionary then
( myObject astype Dictionary  )[ "fourtyTwo" ] = 5 * 9   // or myDictionary = myObject astype
Dictionary
else if myObject istype Vector then
( myObject astype Vector )[ 42 ] = 5 * 9;                // or myVector = myObject astype Vector
```

Because "myObject **astype** Dictionary" is three words forming a group, the Verifier is not always good at seeing "the group" as a unit in its syntax analysis, and you may need to give it some help by enclosing the group in ( ) as above; at other times (as shown in the comments above) the ( ) are not necessary. The same would probably be true if you tried to access ""myObject **astype** Dictionary.Count" where the Verifier might try to access a static **Count** property of the Dictionary class, which doesn't exist, rather than your intention to access **myObject.Count** once you'd convinced the Verifier that **myObject** will have a .**Count** property when the statement is executed even though it doesn't when the program is being verified. Try the statement without, then if the Verifier can't recognise it properly, add brackets.

Collections

Collections, like arrays, allow you to store many values within a single named package. Unlike arrays, the items in a collection don't all have to be the same type, and the collection can be made to expand and shrink depending on how many items they contain, with no maximum size. Like all the objects, only one copy of a specific object exists, shared by all bars, so you can't use [ ] to refer to the collection's state as it was N bars ago. Using a collection rather than an array reduces the RAM consumption compared to the many copies of the array that are retained for history bars. Arrays can't be passed to methods, as far as I have been able to discover, but collections can.

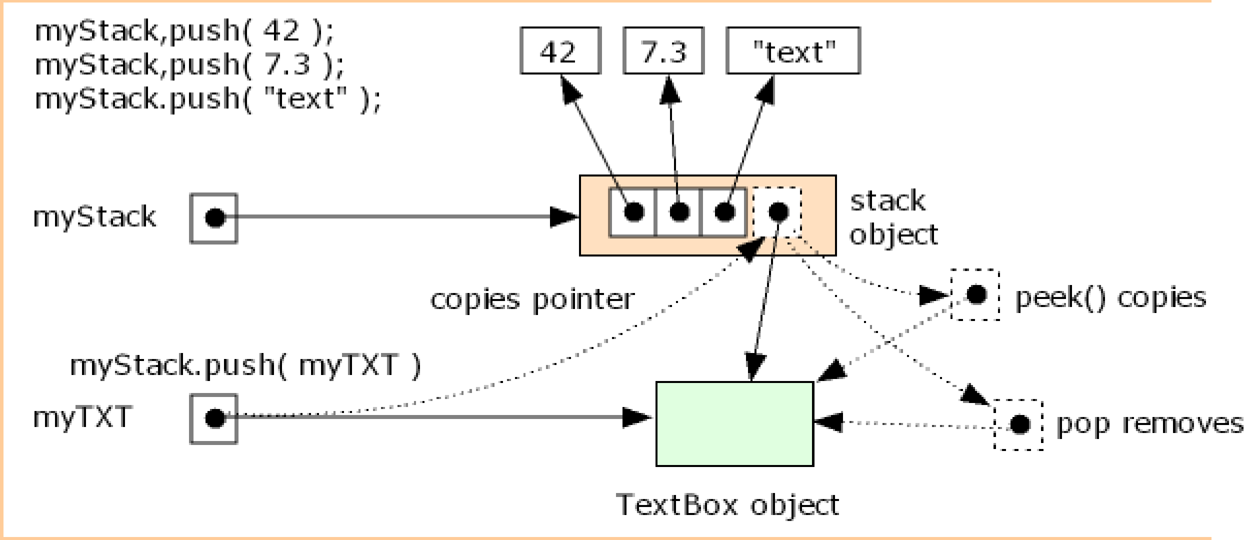
Collections represent linear lists; to get a 2-dimensioned structure or higher, you put collections inside collections. The current options are

- Stack - items are added at one end, only the last item added can be accessed or removed, no other items can be accessed
- Queue - items are added at one end, read or removed at the other end, no other items can be accessed
- Vector - items can be accessed by numerical index, added or removed at the high-index end, or inserted or erased by position
- Dictionary - a list in which items are selected by providing a unique "key string"; items can be added, changed or removed at any position
- GlobalDictionary - a version of a Dictionary that can be accessed from more than one program

A TokenList is not a "collection" in the same sense as the others; it is defined in tsdata.common rather than elsystem.collections, can only hold only strings rather than "anything", and the terminology is different.

Stacks, Queues and Vectors

In .Net everything stored in a collection is stored as an object. This gives the maximum flexibility, but involves a lot of **astype** words when reading items out of a collection. When an object is "added to a collection", it is the pointer to the object that is copied into the collection. When a simple type like an **int**, **double**, **bool** or **string** is added in .Net, the collection seems to create an object to hold the simple value and save the pointer to the created object in the collection; Microsoft calls this "wrapping a value in an object" **boxing**. A stack would then be



```
myStack,push( 42 );
myStack,push( 7.3 );
myStack.push( "text" );
```

I'm not clear whether OOEL collections follow Microsoft's implementation or not. It could be that values like **int** or **double** are actually stored in the collection, to reduce overhead. I haven't been able to devise a test to distinguish whether "myVector.push_back(42)" saves the **int** value 42 in the vector, or creates an object to hold the 42 and saves a pointer to the object in the vector. If anyone can find any evidence one way or the other, please let me know.

The clone process is supposed to create a duplicate of the original collection. When the collection holds externally-created objects, the duplicate collection holds pointers to the same objects as the original collection - changing an object property via one collection means the "copy" sees the same changed property value. However, this doesn't seem to be the same for simple values, as if they are duplicated along with the collection. I tried

```
v1 = New Vector;
v1.push_back( 42 );
v1.push_back( "5 * 9" );
v2 = v1.Clone( ) astype Vector;
v1[ 0 ] = 27;
v1[ 1 ] = "changed";
if v2[ 0 ] istype int then print( "v1[ 0 ] = ", v1[ 0 ] astype int:0:0, " v2[ 0 ] = ", v2[ 0 ] astype
int:0:0 );
if v2[ 1 ] istype string then print( "v1[ 1 ] = ", v1[ 1 ] astype string, " v2[ 1 ] = ", v2[ 1 ] astype
string );
```

which printed

```
v1[ 0 ] = 27   v2[ 0 ] = 42
v1[ 1 ] = changed   v2[ 1 ] = 5 * 9
```

making it look as the values in vectors v1 and v2 are independent rather than v1[0] and v2[0] holding pointers to the same object, but it could just be that the "v1[0] = 27" created a new object to hold the 27 and put that into the vector. Whereas for a genuine object

```
txtA = TextBox.Create( "42", 60, 20 );
v1 = New Vector;
v1.push_back( txtA );
v2 = v1.Clone( ) astype Vector;
txtA.Text = "changed";
if v2[ 0 ] istype TextBox then
        print( "v1[ 0 ] = ", ( v1[ 0 ] astype TextBox ).Text, " v2[ 0 ] = ", ( v2[ 0 ] astype
TextBox ).Text );
```

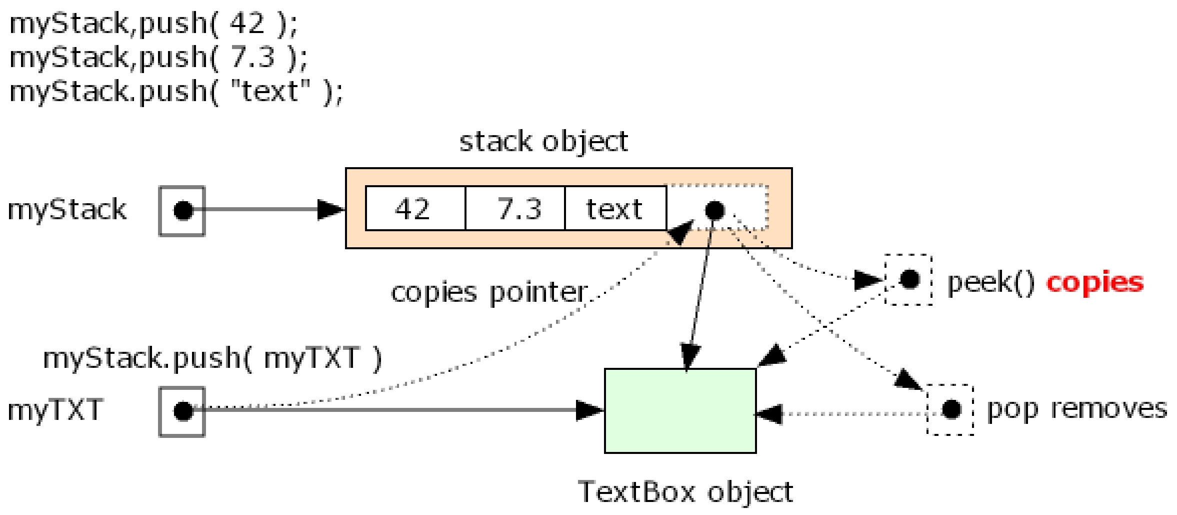
PrintLog shows that the two items are not independent, the clone process creates a copy of the vector and all its pointers, but not the objects that are pointed at, so both vectors contain pointers to a single object

```
v1[ 0 ] = changed   v2[ 0 ] = changed
```

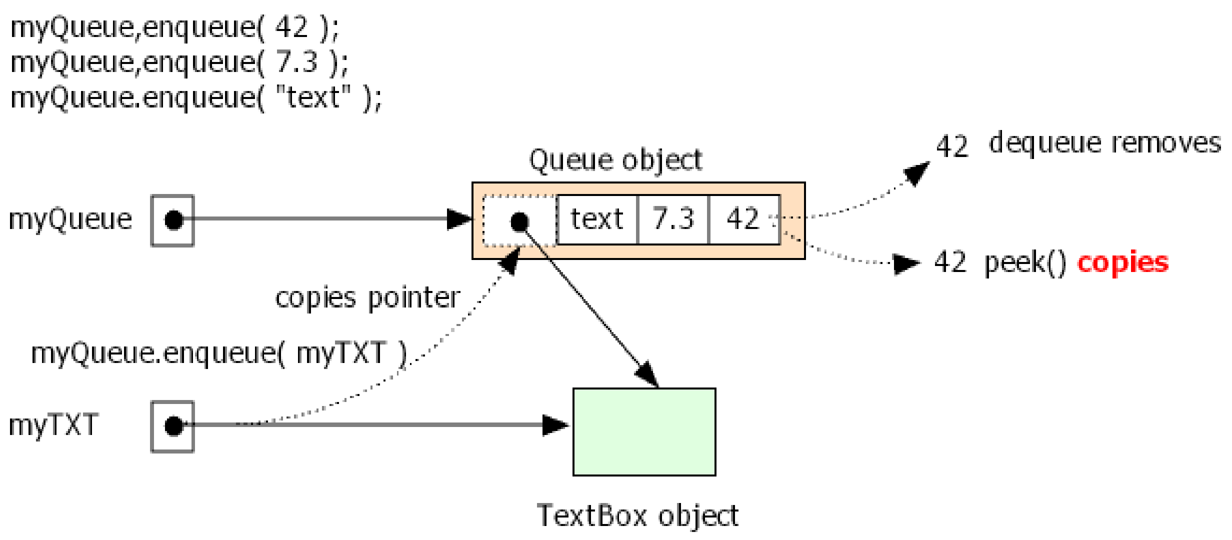
dp2002 pointed out that if you can pass simple values like int and doubles through a GD created with .**Create**(**true**, while you can't pass objects, that implies that ints and doubles are stored in the collection not via pointers.

I'm always puzzled by strings in EL as, like objects, they are too big to fit into a simple variable, and their size can be changed as the program runs. I assume they must be handled in a similar way to objects, but don't know how.

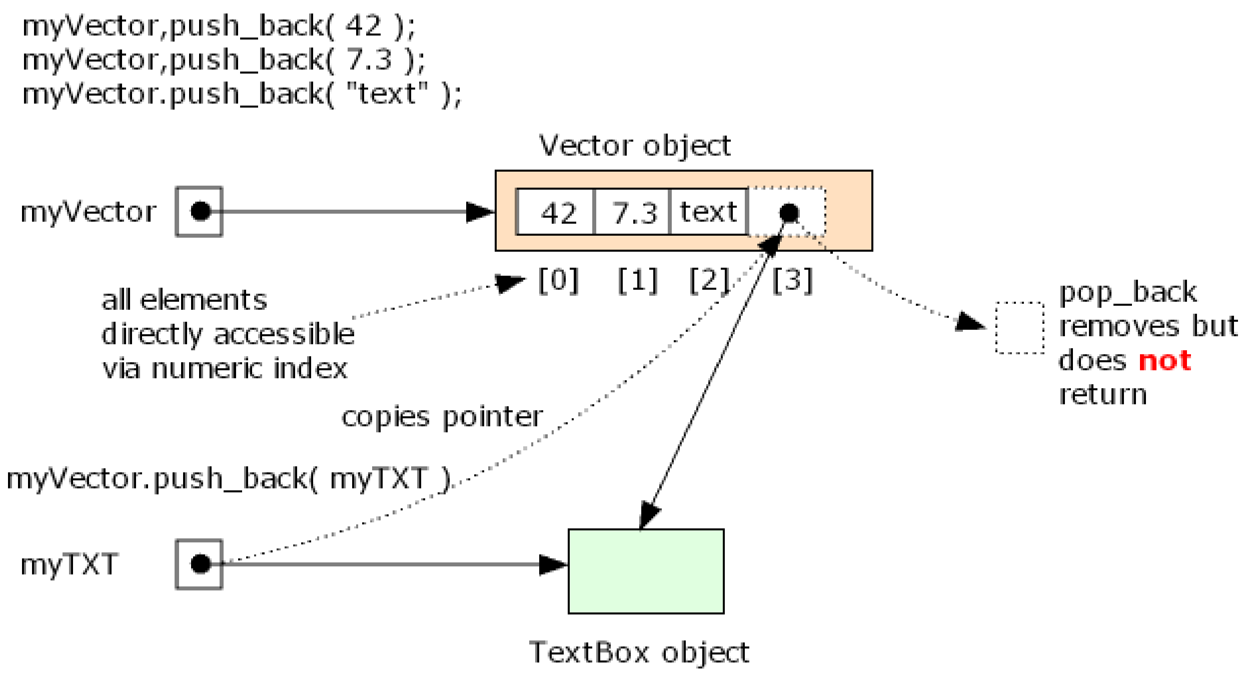
Pending a definite answer to whether simple values are held in the collection in the same way as objects,, I'll carry on with the assumption that simple values (ints, doubles, bools and strings) are stored inside the collection object, the collection can also hold pointers to objects I create, but to keep the syntax consistent the Verifier requires you to **always** specify an **astype** when reading a value out of a collection. If nothing else, it makes the diagrams easier, so a stack is



Queue :



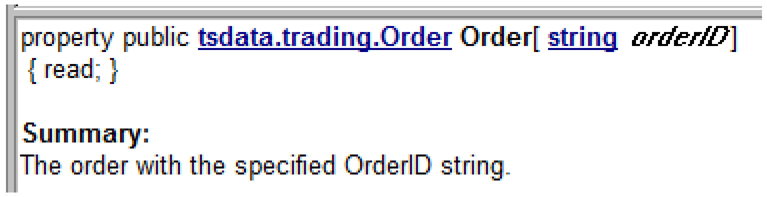
Vector :


myVector,push_back(42);
myVector,push_back(7.3);
myVector.push_back("text");
Vector object
myVector
42
7.3
text
[0]
[1]
[2]
[3]
all elements directly accessible via numeric index
pop_back removes but does not return
copies pointer
myVector.push_back(myTXT)
myTXT
TextBox object

Many of the TS classes incorporate collections, but without needing **astype**, either because access to the collections is provided by **indexer** functions that return a specific type of object, or (possibly) because the are built on collections that inherit their basic functionality from the general collections by extend them by restricting the contents to objects of only a certain type. For instance

print(MyOrdersProvider.**Order**["1-234-5678"].Symbol);

doesn't need **astype**, because the **Order** indexer returns an object of type **Order** not of type **Object**, as shown in the TDE Dictionary


property public tsdata.trading.Order Order[string *orderID*]
{ read; }

Summary:
The order with the specified OrderID string.

For the purpose of making it easier to convert array-based programs to Vector-based, eg to use with methods, I've suggested that TS provide classes intVector and doubleVector to hold only values of those types (and which allow an initial size to be provided as a parameter to their .Create methods) but I doubt that it will happen - excluding me, 2 votes of "worth doing" and one of "not important". At least I didn't get the "negative affect" vote that seems to be traditional.

Stack, Queue, Vector, and Dictionary operate on the same basic principles, as does the GlobalDictionary under some circumstances; when a GlobalDictionary is sharing between, say, a chart and RadarScreen, or when multi-core is enabled, it could not hold objects before Update 17, and in the current beta of Update 17 (Feb 2103) objects can be added but the mechanism has a subtle difference. TokenLists, being specific to strings, is the odd one out.

In writing test programs for this document, I discovered that the **for** loop doesn't change the number of repetitions if you modify the terminal value from within the loop.

```
j = 5;
For index = 0 to j begin
print( index:0:0 );
j = j - 1;
end;
```

prints 0, 1, 2, 3, 4, 5 - not 0, 1, 2. The terminal value is saved on entry to the loop and becomes independent of the expression after the **to**. You don't need to worry about looping from 0 to Count if the processing in the loop changes the value of Count.

Interfaces

An "**item**" can be a simple value (**int**, **double**, **bool**, **string**) or an object,

Stack, Queue, Vector, Dictionary	Clear( )	removes all items from the collection
	object Clone( )	"creates a duplicate" of the collection, including all the

		pointers it contains; it **does not** appear to **clone the objects** that those pointers point at; treat the result with **caution** needs **astype** on the returned object
	int Count	the number of items in the collection
	object Create( )	creates an object from the specified class, returns a pointer to the object, of the right type (Stack, Queue, Vector, or Dictionary), **no need for astype**
Stack and Queue	**object** peek( )	returns a pointer to the item that would be removed by the removal method pop/dequeuue; needs **astype**
Stack	push(item)	add a new item on the end of the list
	object pop( )	removes the end item from the list, and returns it as a result; needs **astype**
Queue	enqueue(item)	add a new item on the front of the queue
	object dequeue( )	removes the back item from the queue, and returns it as a result; needs **astype**
	bool contains(item)	returns true if the item is in the queue; item is a simple value or an object variable; eg q.contains("one"), q.contains(myString), q.contains(myTextBox)
Vector	**object** at(index)	returns the item at **int** position **index**; I don't see the point of this, compared to **[** index **]**; needs **astype**
	object back( )	returns a copy of the last item in the vector, but does not remove it; needs **astype** **pop_back** removes the last item, but does **not** return it as a result - go figure
	bool empty( )	returns true if there are no elements (ie .Count = 0), false otherwise
	int erase(index)	removes the item at **int** position index (and returns the value of index; I wonder why?)
	int erase(first, last)	removes the items from **int** position first up to, **but not including**, the item at **int** position last returns the value of first as a result, which will be the position of the first item that was not removed will be; if you want to remove from an item to the end, **inclusive**,

	use
	myVec.erase(42, myVec.Count - 1); // .... , myVec.Count) raises an exception
	myVec.erase(42);
	or, maybe, myVec.erase(myVec.erase(42, myVec.Count - 1)); using the returned value?

object front( )	returns the first item in the vector, ie myVec[**0**]; needs **astype**
insert(index, item)	inserts the new item (value or object) at **int** position index and moves the rest up one space it doesn't allocate more RAM to extend the vector if index >= count, except if count = 0 & index = 0 insert(0 when the vector is empty works, insert(2 when it has 2 items **doesn't** work
insert(index, item, N)	inserts the new item (value or object) at **int** position index N times, and moves the rest up
pop_back( )	removes the last item from the vector, but doesn't tell you its value - read it before pop_back( )
push_back(item)	adds a new item on the end of the vector, extending the vector's RAM allocation

Suppose you wanted to write a program that placed and tracked orders, using **Order** objects. The object has a **string** property **OrderID** by which it is uniquely identified. But the user can place orders manually alongside the program, and you wanted the program to only respond to its own orders. A **Vector** isn't the best collection for this, but if you had decided to use one then you could save **OrderID**s as the program generated orders by

```
once myOrders = new Vector;
...
myOrders.push_back( newOrderID );      // once the program has placed a new order and
obtained the OrderID
```

and when the program notices that something has happened to a specific order it could check whether it is one it should pay attention to by

```
Method bool mine( string ID )
vars:
int count;
begin
for count = 0 to myOrders.Count - 1 begin
```

```
if ID = myOrders[ count ].OrderID astype string then return true;
end;
return false;
end;
```

allowing

```
if mine( newOrderID ) then .....
```

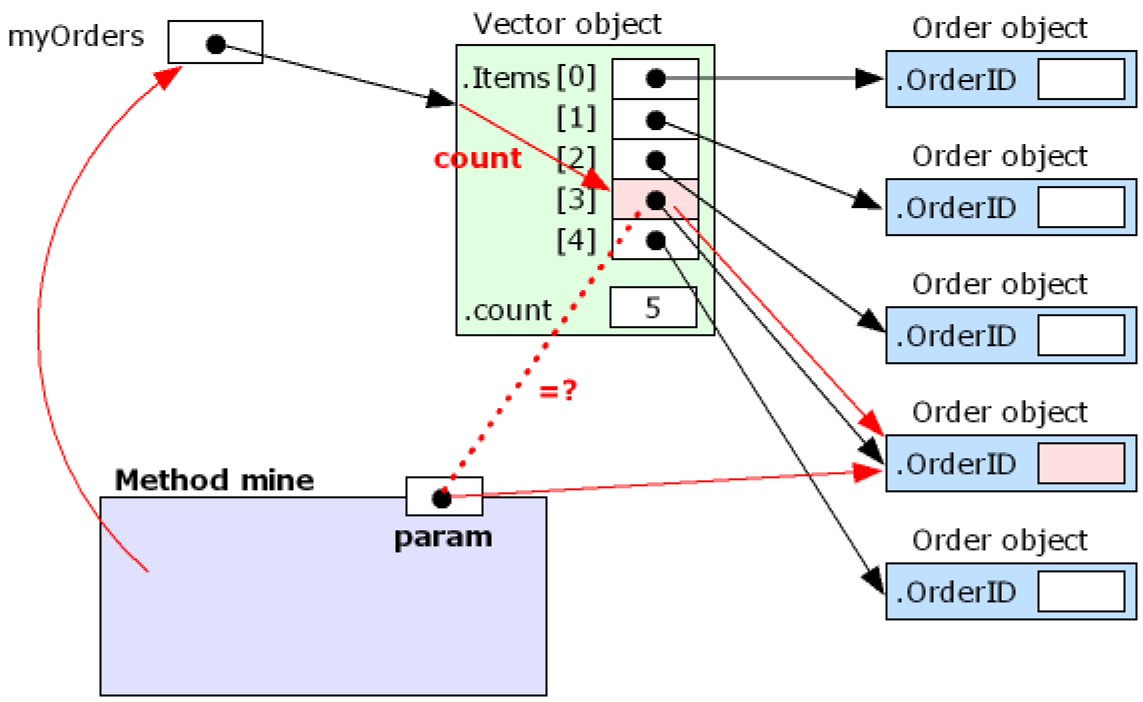
This is no different to having an array of strings, really, except that it is an array that is always "just the right size". But because a **Vector** holds objects, an alternative would be to save actual **Order** objects in the Vector, rather than just ID strings, so that the program could check on any properties of only its own orders :

```
once myOrders = new Vector;
...
myOrders.push_back( newOrderObject );     // once the program has placed a new order and
obtained the OrderID
```

and

```
Method bool mine( Order param )
vars:
int count;
begin
for count = 0 to myOrders.Count - 1 begin
if param = myOrders[ count ] astype Order then return true;
end;
return false;
end;
```

Here, the if statement is testing whether the parameter is a pointer to the same object as one of the items in the vector, rather than comparing a property of one of the vector items.



This version of the method gets access to the **Vector** object by following the pointer from the non-local variable **myOrders**, uses is local **count** variable to select one of the entries in the **Items** collection inside the object, then compares the pointer if finds in that item with the pointer it was given as its parameter to see if they are actually pointing at the same object. If it wished to it could follow the pointer from the **Vector** to an actual **Order** object, and access the .**OrderID** property stored inside the **Order** object.

When experimenting with collections, you are likely to want to be able to display the contents of the collection. For a **Vector**, you could try

```
Method void printVector( Vector param )
Vars:
string prefix,
int n;
begin
If param = Null then
print( "Vector is null" )
Else if param.Count = 0 then
print( "Vector is empty" )
Else begin
For n = 0 to param.Count - 1 begin        // scan all items in the vector, identifying its type and
displaying it as appropriate
prefix = "[ " + Numtostr( n, 0 ) + " ] is ";
If param[ n ] = Null then
```

```
print( prefix, "null" )
Else if param[ n ] istype int then
print( prefix, "an int = ", param[ n ] astype int:0:0 )
Else if param[ n ] istype double then
print( prefix, "a double = ", param[ n ] astype double:0:5 )
Else if param[ n ] istype bool then
print( prefix, "a bool = ", param[ n ] astype bool )
Else if param[ n ] istype string then
print( prefix, "a string = '", param[ n ] astype string, "'" )
Else
print( prefix, "an object of type ", param[ n ].GetType( ).ToString( ) );     // see below
end;
end;
end;
```

which, in a program

```
v1 = New Vector;
v1.push_back( 42 );
v1.push_back( 54.0 );
v1.push_back( true );
v1.push_back( "life, the universe and everything" );
v1.push_back( txtA );
v1.push_back( s1 );
v1.push_back( q1 );
v2 = v1.Clone( ) astype Vector;
v1.push_back( v2 );
print( "Vector v1 :" );
printVector( v1 );
```

gives

```
[ 0 ] is an int = 42
[ 1 ] is a double = 54.00000
[ 2 ] is a bool = TRUE
[ 3 ] is a string = 'life, the universe and everything'
[ 4 ] is an object of type elsystem.windows.forms.TextBox
[ 5 ] is an object of type elsystem.collections.Stack
[ 6 ] is an object of type elsystem.collections.Queue
[ 7 ] is an object of type elsystem.collections.Vector
```

getType( ) is a method that all objects inherit from the generic **Object** class, that basically represents the class from which the object is created; it isn't represented as a text string, so you have to call its .**ToString**( ) method to get something printable.

If you wanted to experiment with vectors inside vectors you could extend the method by making it **recursive** (calls itself)

```
Method void printVector( Vector param, int indent )
    ....
    prefix = spaces( indent ) + "[ " + Numtostr( n, 0 ) + " ] is ";
    ....
    Else if param[ n ] istype Vector then begin
        print( prefix, "a vector = " );
        printVector( param[ n ] astype Vector, indent + 5 )
```

Then

printVector(v1, 0);

gives

```
.....
[ 7 ] is a vector =
[ 0 ] is an int = 42
[ 1 ] is a double = 54.00000
[ 2 ] is a bool = TRUE
[ 3 ] is a string = 'life, the universe and everything'
[ 4 ] is an object of type elsystem.windows.forms.TextBox
[ 5 ] is an object of type elsystem.collections.Stack
[ 6 ] is an object of type elsystem.collections.Queue
```

Note that **v1** was cloned to produce **v2** before a vector was added to **v1** so **v2** does not contain a vector; v2 can contain vectors, which can contain vectors ... but if **v2** contained a reference to **v1** somewhere in the depths, and **v1** contains a reference to **v2**, this process would go into an endless loop as printing **v2** would start printing **v1** from the beginning in the middle of printing **v1**, Recursion can be helpful, but getting it wrong is lethal.

Printing a queue's contents is a bit more complex, because only one item within the queue is visible. The easiest way is to rotate the queue by one position, for as many times as it contains items, which will print in the order items will leave the front of the queue.

```
Method void printQueue( Queue q )
vars:
Object item,
int index;
begin
for index = 0 to q.Count - 1 begin
```

```
item = q.Dequeue( );        // note the item is a generic object at this stage, later to be tested
with isType.
q.Enqueue( item );
.... print item in the same way as the vector example
.... or just print( item.ToString( ) );
end;
end;
```

A Vector offers the same functionality as a stack, except that its **pop_back**( ) returns **void**, so to emulated a stack's pop( ) method you need to use

```
item = myVector.back( );
myVector.pop_back( );
```

so I'm not sure if there's a compelling case for using a **Stack** object, But if you did, you could print its items in the order they would come off the stack by

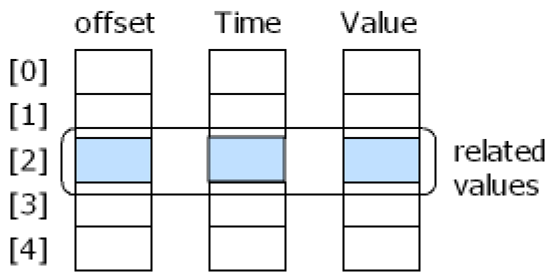
```
Method void printVector( Vector vec )
vars:
Object item,
Vector temp,     // duplicate of the parameter, to be destroyed during printing
int index;
begin
temp = vec.clone( );                    // duplicate the parameter, as the read process is
destructive
for index = 0 to vec.Count - 1 begin        // terminal value is saved when the loop starts, so
changing it within the loop has no effect
item = temp.pop( );                 // remove the last item from the collection.
.... print item in the same way as the vector example
.... or just print( item.ToString( ) );
end;
end;
```

Suppose you wanted to scan for high pivots in recent history, defining a pivot as being a High preceded by **leftStrength** bars less than the high, and followed by **rightStrength** bars also less than the high. You could return a list of bar offsets to such pivots in a vector by

```
Method Vector highPivots( int length, int leftStrength, int rightStrength )
vars:
bool isPivot,         // during checking, set to false if offset isn't a pivot
int offset,           // selects a bar that might be a pivot
int index,            // use as an offset from offset when looking for bars that show [ offset ] isn't
a pivot
```

```
double candidate,    // the high value of bar [ offset ]
Vector answer;       // vector showing where pivots were found
begin
answer = New Vector;
offset = Rightstrength + 1;
If BarNumber < leftStrength + offset then return answer;     // return an empty vector if a pivot
is impossible
While offset <= length - leftStrength begin                   // scan back through history checking
for pivots
candidate = high[ offset ];                        // assume that [ offset ] is a high pivot
isPivot = True;
index = 1;
While isPivot and index <= rightStrength begin                 // check if any bars to the right
show [ offset ] isn't a pivot
If High[ offset - index ] >= candidate then isPivot = false;    // no, it isn't a pivot, exit the loop
index += 1;                                        // increment "distance from offset" by 1
end;
index = 1;
While isPivot and index <= leftStrength begin                  // scan bars to the left to see if
they show [ offset ] isn't a pivot
If High[ offset + index ] >= candidate then isPivot = false;  // no, it isn't a pivot, exit the loop
index += 1;                                        // increment "distance from offset" by 1
end;
If isPivot then answer.push_back( offset );                  // if neither checks show
[ offset ] is not a pivot, save it
offset += 1;                                              // advance by 1 to next candidate
end;
return answer;                                               // return a vector holding
offsets to each pivot found
end;
```

Now suppose that you wanted to return more than just offsets from such a method - say you also wanted the time and value of the pivot. One approach would be to return multiple vectors, where values at the same index relate to each other



This could be achieved via two extra vector parameters :

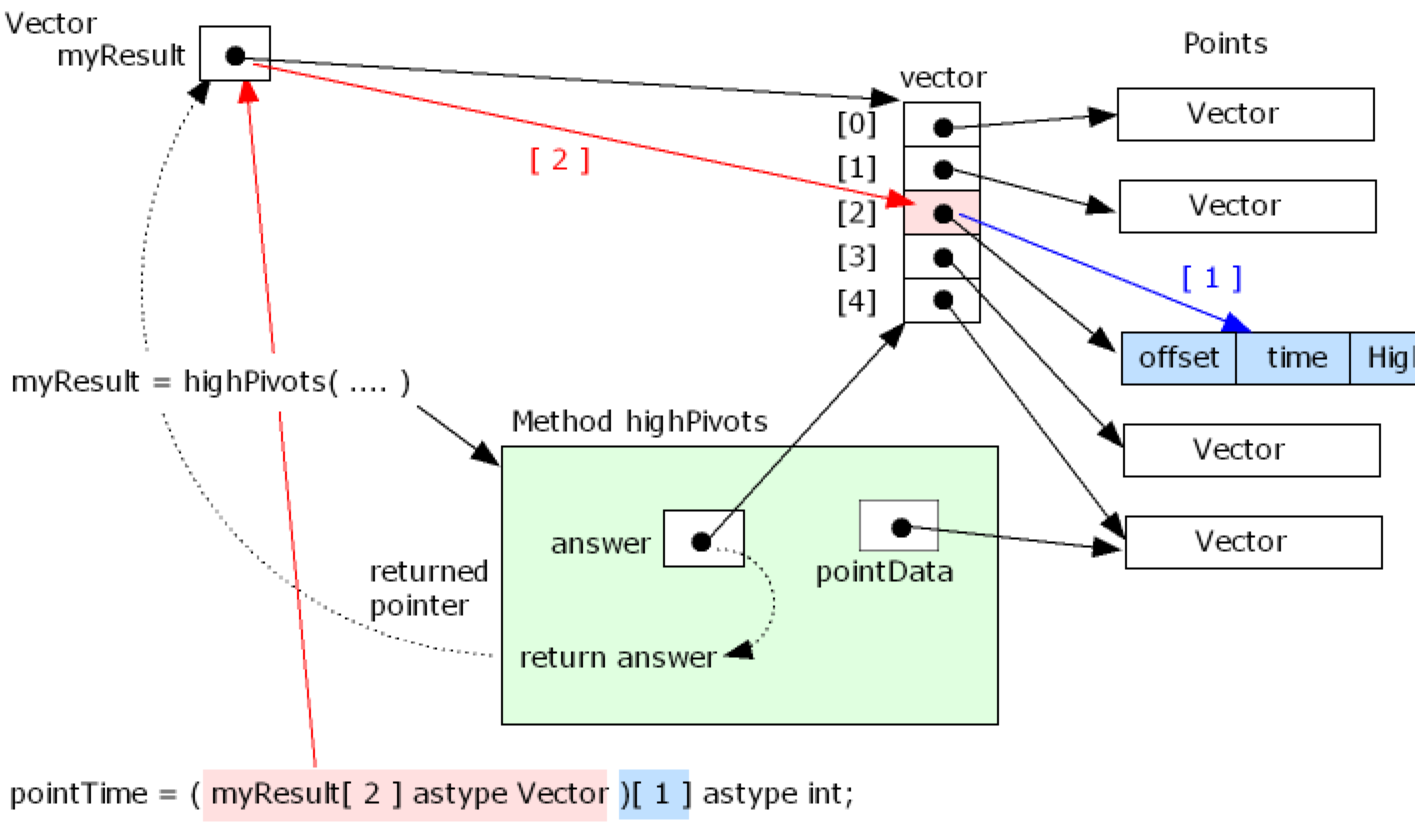
```
Method Vector highPivots( int length, int leftStrength, int rightStrength, out Vector pivotTime,
out Vector pivotHigh )
....
pivotTime = new Vector;
pivotHigh = new Vector;
....
If isPivot then begin
        answer.push_back( offset );
        pivotTime.push_back( time[ offset ] );
        pivotHigh.push_back( high[ offset ] );
        end;
```

The **out** prefixes are needed not because the method will be putting values into the two vectors, but because the method creates the vector objects and needs to pass the pointers to the vectors back into the caller's variables.

A second possibility would be to bind the three values related to a pivot into a vector, then add that vector to **answer**, so that the return "value" from the method contains all the information.

```
Method Vector highPivots( int length, int leftStrength, int rightStrength )
vars:
bool isPivot,          // during checking, set to false if offset isn't a pivot
int offset,            // selects a bar that might be a pivot
int index,             // use as an offset from offset when looking for bars that show [ offset ] isn't
a pivot
double candidate,      // the high value of bar [ offset ]
Vector pointData,      // details of a single pivot point
Vector answer;         // vector showing where pivots were found
begin
answer = New Vector;
offset = Rightstrength + 1;
If BarNumber < leftStrength + offset then return answer;     // return an empty vector if a pivot
is impossible
While offset <= length - leftStrength begin                     // scan back through history checking
for pivots
candidate = high[ offset ];                             // assume that [ offset ] is a high pivot
isPivot = True;
index = 1;
While isPivot and index <= rightStrength begin                         // check if any bars to the right
show [ offset ] isn't a pivot
```

```
If High[ offset - index ] >= candidate then isPivot = false;     // no, it isn't a pivot, exit the loop
index += 1;                                              // increment "distance from offset" by 1
end;
index = 1;
While isPivot and index <= leftStrength begin                       // scan bars to the left to see if
they show [ offset ] isn't a pivot
If High[ offset + index ] >= candidate then isPivot = false;   // no, it isn't a pivot, exit the loop
index += 1;                                              // increment "distance from offset" by 1
end;
If isPivot then begin
pointData = new Vector;                                  // create a vector to hold details of a single
pivot point
poiutData.push_back( offset );                           // add the point's offset, time and value to
the point vector
poiutData.push_back( time[ offset ] );
poiutData.push_back( High[ offset ] );
answer.push_back( pointData );                            // add the point's vector to the vector
that will be returned
end;
offset += 1;                                                  // advance by 1 to next candidate
end;
return answer;                                                        // return a vector holding
offsets to each pivot found
end;
```



1. When the main program executes **myResult = highPivots(....)** control transfers to the **highPivots** method.
2. This creates a **Vector** and puts a pointer to it in the local variable **answer**
3. When the method finds the first high pivot is
    a. creates a new vector and puts a pointer to it in local variable **pointData**
    b. adds three values to the vector
    c. copies the pointer to the vector on to the end of the vector pointed at by **answer**
4. Each time the method finds another high pivot, it repeats the steps in 3. above. Using the same variable **pointData** for each new vector created doesn't destroy any values, because by the time **pointData** is re-used the vector object(s) it used to point at are kept alive by the pointers to them that were stored in **answer**, In the diagram, **pointData** would have been re-used for four more vectors after the first was created.
5. When control leaves the method, the pointer from **answer** is copied back to the caller, who saves it in a variable which will then be used to access the whole collection of structured data created by the method.
6. The main program can then access a specific data field for a specific pivot point by a statement like

    pointTime = (myResult[2] **astype** Vector)[1] **astype int**;

In this,

a. the "**myResult**[2]" follows the pointer from variable **myResult** to an object which the declaration of **myResult** promises will be a vector
b. the pointer from item [2] of that vector is read, and the Verifier is told that this will be a pointer to a Vector
c. the pointer is followed and item [1] of that vector is read, and the Verifier is told that the item will, when the program runs, be an **int** value

Using numbers like [2] and [1] in statements that access items in collections works, but isn't very self-explanatory. I believe that the easier it is to understand the text of a program, the more chance there is that the program will be correct. This is especially true when coming back to modify a program a long time after it was written, when you've forgotten all the tricks that seemed obvious while you were doing the original writing.

EL allows you to declare names for constant numbers (sadly, not for strings or bools, at present). This allows you to use the name to convey the purpose of the number in a more self-documenting way. The **Const**: keyword is much like the **Vars**: keyword (in the main program, but not within a method), except that the value in brackets isn't an initial value to be placed in RAM, its a value to be placed in the program in place of the name that represents it. Thus after

```
Const:
int offsetField( 0 ),
int timeField( 1 ),
int highField( 2 );
```

then

```
pointTime = ( myResult[ 2 ] astype Vector )[ timeField ] astype int;
```

works just as well as

```
pointTime = ( myResult[ 2 ] astype Vector )[ 1 ] astype int;
```

but a programming mistake

```
pointTime = ( myResult[ 2 ] astype Vector )[ offsetField ] astype int;
```

stands out more than the equivalent

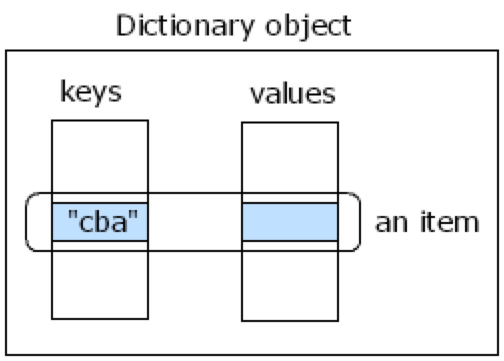
```
pointTime = ( myResult[ 2 ] astype Vector )[ 0 ] astype int;
```

I think.

Dictionaries

The basic principles of the Dictionary and GlobalDictionary are the same, although some of the details differ. The Dictionary is available within a program, the GlobalDictionary makes its contents visible to more than one program, and has events to support that role.

A specific item in a dictionary is selected by providing a "**key** string" value that is **unique** within the dictionary, rather than a numeric index. Items are held internally in the alphabetic order of their associated keys. You can loop through all the items, by accessing two vectors inside the dictionary.



Dictionary Interface

Add(key, value)	Adds an item to the dictionary; **key** is a unique string, **value** can be an object or simple value. Not essential, unless you want to be sure not to overwrite an existing item, as myDictionary[**key**] = **value** will add a new item if there is no match for key in the dictionary (and overwrite an item if there is a match). Add( ) will raise an exception if you try to add an item with a key that is already present in the Dictionary.
Clear( )	removes all elements from the dictionary
object Clone( )	creates a duplicate of the object and its contents; not available for GlobalDictionary; needs **astype**
bool Contains(key)	returns **true** if the **string key** is already in use in the dictionary
int Count	the number of items in the dictionary
Dictionary Create( )	creates a new Dictionary object, and returns a pointer to it GlobalDictionary has a second format that accepts two parameters (**bool**, **string**) to set how widely visible its contents are, and the name by which the

	shared data structure is known to other programs
object Items[key]	the default indexer method that accepts a string parameter and returns a pointer to the item that key belongs to; **case sensitive** Being an indexer, its parameter is enclosed within [ ] rather than ( ); being the default indexer, it will be called if you simply put **[key]** immediately after the name of a Dictionary
Vector Keys	vector holding all the key strings in use in the Dictionary, in alphabetic order; both the **keys** collection **and** its **contents** are **read-only** - you can't either myDict.Keys.push_back("spoiler"), or myDict.keys[0] = "aardvark"; (although you don't get any error messages if you try, the attempts just get ignored)
Remove(key)	removes the item whose key string matches the **string** parameter
Vector Values	vector holding the item values in the dictionary, in the same order as their strings are in .keys; both the **keys** collection **and** its **contents** are read-only, you have to use the myDict[myKey] = ...; format to change a value in Values.

A Dictionary is good for gathering related values together, using names to identify items without the need to define **Consts**. You don't need to code loops to find a specific item in the dictionary, you can easily check if the item is present, and go straight to it if it is. You can't/don't have to sort the items in a dictionary, but you can't insert a new item before a specific existing item (except by an appropriate algorithm for generating unique key strings). You do need to be able to construct a string from an object, which will be unique to that object.

However, the key string matching is **case sensitive** (and you can't define a string **Const**), so a proper spelling error or typing a key including the wrong case will cause a generic exception "Object reference is not set to an instance of an object" rather than a more helpful "Key string not found in Dictionary".

You can protect yourself from these errors, and also from accidentally overwriting an item when intending to add a new item, using .**Contains**( ); eg

```
if myOrders.Contains( myOrderID ) then
print( "Attempt to insert the same order twice" )
else
myOrders[ myOrderID ] = ....;
```

or

```
if myOrders.Contains( myOrderID ) then
print( ( myOrders[ myOrderID ] astype Order ).Symbol )
```

```
else
print( ( myOrderID, " not found" );
```

Using a Dictionary to filter order events, so that only ones from Orders issued by the program are dealt with, is quite easy. You'd use the **OrderID** as the unique string, so the list would be built up by, for instance

```
Dictionary myOrders( Null );
...
once myOrders = new Dictionary;
...
myOrders[ myOrderID ] = myOrder;                // save the actual order object in the filter list, but
the "value" of an item could be anything
...
if myOrders.Contains( newOrderID ) then .....    // process an order if its ID is in the Dictionary
```

A Dictionary maintains a sorted order for its keys, although it is alphabetical order rather than with a numerical order option, so "123" comes before "9". You don't have to distinguish between "insert" and "overwrite" if you don't want to. You can treat a Dictionary as a pseudo user-defined object, treating the key strings as property names; I've used Dictionaries to represent a trade by

```
myTrade = new Dictionary( );
myTrade[ "entry" ] = myEntryOrder;
myTrade[ "target" ] = myTargetOrder;
myTrade[ "stop" ] = myStopLoss;
```

then all three orders can be passed as a parameter to a method by passing the **myTrade** dictionary. Using a dictionary to represent a single pivot point, the highPivots method above could be :

```
Method Vector highPivots( int length, int leftStrength, int rightStrength )
vars:
bool isPivot,              // during checking, set to false if offset isn't a pivot
int offset,                // selects a bar that might be a pivot
int index,                 // use as an offset from offset when looking for bars that show [ offset ]
isn't a pivot
double candidate,          // the high value of bar [ offset ]
Dictionary pointData,      // details of a single pivot point
Vector answer;             // vector showing where pivots were found
begin
answer = New Vector;
offset = Rightstrength + 1;
If BarNumber < leftStrength + offset then return answer;     // return an empty vector if a pivot
```

```
is impossible
While offset <= length - leftStrength begin                    // scan back through history checking
for pivots
candidate = high[ offset ];                              // assume that [ offset ] is a high pivot
isPivot = True;
index = 1;
While isPivot and index <= rightStrength begin                         // check if any bars to the right
show [ offset ] isn't a pivot
If High[ offset - index ] >= candidate then isPivot = false;     // no, it isn't a pivot, exit the loop
index += 1;                                               // increment "distance from offset" by 1
end;
index = 1;
While isPivot and index <= leftStrength begin                         // scan bars to the left to see if
they show [ offset ] isn't a pivot
If High[ offset + index ] >= candidate then isPivot = false;   // no, it isn't a pivot, exit the loop
index += 1;                                               // increment "distance from offset" by 1
end;
If isPivot then begin
pointData = new Dictionary;                               // create a vector to hold details of a single
pivot point
poiutData[ "offset" ] = offset;                            // add the point's offset, time and value to
the point vector
poiutData[ "time" ] = time[ offset ];
poiutData[ "high" ] = High[ offset ];
answer.push_back( pointData );                             // add the point's vector to the vector
that will be returned
end;
offset += 1;                                                  // advance by 1 to next candidate
end;
return answer;                                                       // return a vector holding
offsets to each pivot found
end;
```

and the caller would now access the time field for pivot [2] by

```
pointTime = ( myResult[ 2 ] astype Dictionary )[ "time" ] astype int;
```

If you wanted to print the contents of a dictionary :

```
Method void printDictionary( Dictionary param )
vars:
int index,
string keyIdent,
string key;
Begin
If param = Null then print( "Dictionary is null" )
Else if param.Count = 0 then print( "Dictionary is
empty" )
Else begin
print( "Printing dictionary" );                    // scan all the keys of dictionary items
For index = 0 to param.Count - 1 begin             // retrieve the next key
key = param.Keys[ index ] astype string;           // identify the key
keyIdent = "[ " + key + " ] = ";                   // or the series of "if param[ key ] istype int
print( keyIdent, param[ key ].ToString( ) );       then .....
end;
end;
end;
```

TokenList

A tokenlist is a specialised collection (of strings, not objects) listed in the **tsdata.common** section in the TDE Dictionary. It seems a bit quirky, so I suspect it was invented for internal use within TS, eg for lists in the **Provider** objects, with specific needs then allowed to become public. It is basically "a comma-separated list of words" (where a "word" is any sequence of characters that doesn't include a comma, I think), but with vector-like direct access to specific numbers in the list.

The role of leading and trailing spaces for a word in the list is puzzling, in that they are at times preserved when you look at the whole list (ToString( ) or Value), but when you read just an individual word using [ ] such spaces are not returned.

```
tkn1 = tokenlist.Create( );                        // create without initialisation

tkn = tokenlist.Create( "one,two,     three    ,                  one" ); // create & initialise

print( newline, "Value = ", tkn.Value );

print( "ToString = ", tkn.ToString( ) );

print( "[ 1 ] = '", tkn[ 1 ], "'" );       // uses the default indexer item to read a word from the list

print( "[ 2 ] = '", tkn[ 2 ], "'" );

print( "[ 3 ] = '", tkn[ 3 ], "'" );

tkn.add( "      new     " );             // extend the list using the add method

tkn += "   one";                        // extend the list using the += operator
```

```
tkn.Insert( 2, "new" );                    // insert into the body of the list

print( tkn.ToString( ) );
```

gives, in PrintLog

```
Value = one,two,     three    ,                  one

ToString = one,two,     three    ,                  one

[ 1 ] = 'two'

[ 2 ] = 'three'                                   // note spaces have been stripped

[ 3 ] = 'one'

one,two,new,three,one,new,one
```

although I have seen the extra spaces preserved in ToString() for longer than the above, but that may be a glitch; I have't seen spaces preserved in reading an individual word.

There is no restriction on having the same word in the list more than once. IndexOf and removal seem to be case-insensitive, unlike Dictionaries.

TokenList Interface

Add(string)	Adds the string on the end of the list
Clear( )	Removes all items from the list
object Clone( )	Creates a duplicate of the object; needs **astype**
bool Contains(string)	returns true if the string exists as an item in the list; case insensitive
int Count	the usual "number of items in the collection"
TokenList Create	creates a TokenList object, returns a pointer to it
TokenList Create(string)	creates a TokenList object, returns a pointer to it, and populates it from the comma-separated list of values provided as a parameter. Spaces before and after a comma seem to be preserved in the list, but are stripped when one word is read out
int IndexOf(string)	returns the index of the first occurrence of the string parameter in the list, or -1 if not found; match is case insensitive
Insert(int, string)	"Help" says "Inserts the specified index after the specified name." text is wrong. It inserts the string at position int in the list, and moves everything right to make space

bool IsReadOnly	a read-only flag showing if the **TokenList** is read-only, but I see no method of turning "read-only" on or off. Odd.
string Item(**int**)	indexer that picks a specific word out of the list, removing leading and trailing spaces. Being the **default indexer**, it makes it possible to access a word simply by putting an **int** index in **[ ]** after the name of the TokenList object
Remove(string)	removes the specified item from the list; case insensitive, exception if not found
RemoveAt(int)	removes the item at a specific position in the list; exception if not found
string ToString( )	returns the whole list as a single string
string Value	same as ToString?
+= operator	adds the string on the right to the end of the **TokenList** object on the left
-= operator	removes the first instance of the string on the right from the **TokenList** object on the left; case insensitive

Possibly the best candidate for "filtering out only the program's own orders";

```
once nyOrders = TokenList.Create( );
...
myOrders += newOrderID;
....
if myOrders.Contains( anOrderID ) then ..... // process it as one of my orders
```

Events

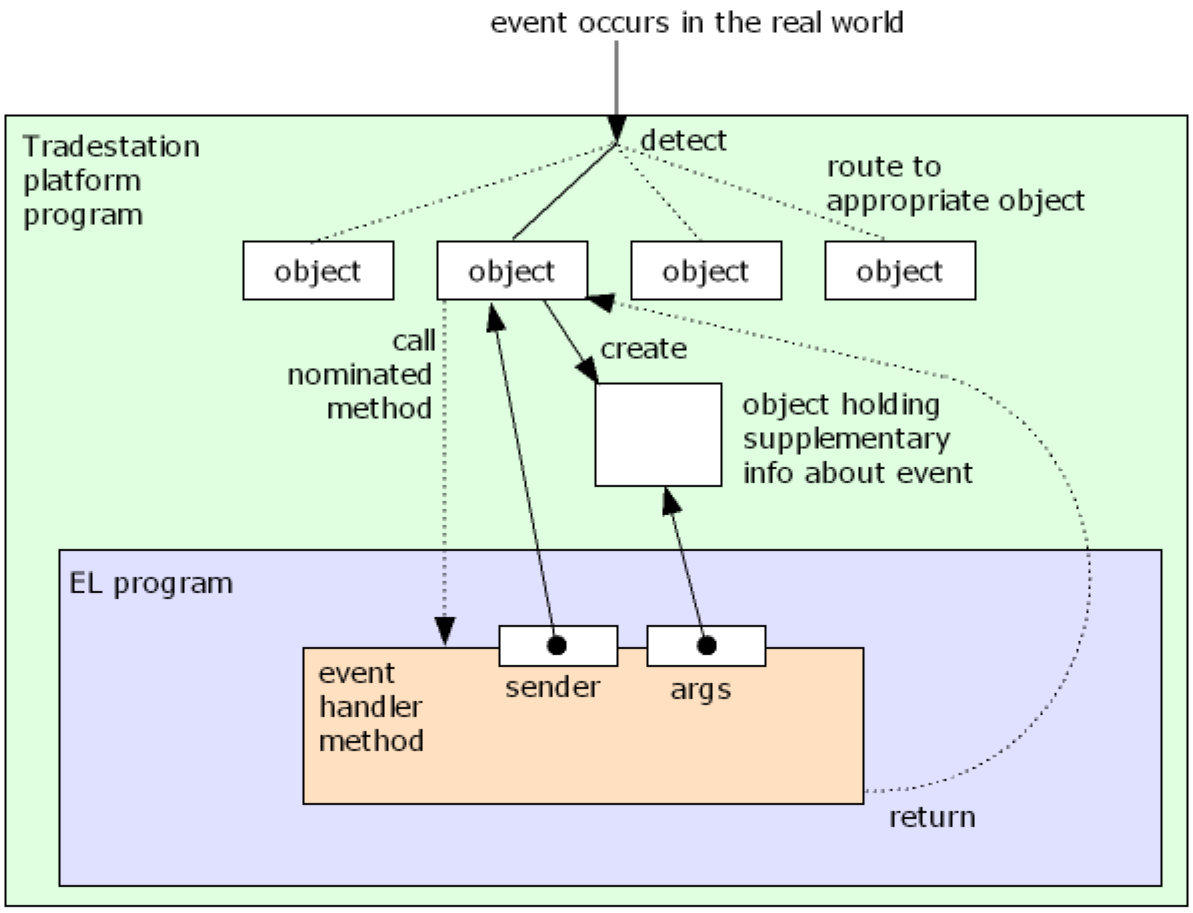
The executable form of an EL program is contained within the TS platform program. The platform program is in contact with the real world (mouse, screen remote servers, etc). When the platform sees that "something has happened" in the real world, that a specific program needs to know about, the platform is able to execute a method within the EL program, so that the EL program has the chance to do something about whatever happened. The "something has happened" message is an **event**, and the method that is called is an "**event handler**".

The event handler method is just a normal method coded in the EL program, but the code to call that event handler is already embedded in the TS platform program, so you have no freedom over the type or number of parameters that the event handler will receive (although you can chose your own names if you wish, as they are local to the handler); they are fixed by the TS platform code :

1. an object representing the source of the event (an order, a form control, or whatever)

2. an object that can contain supplementary information about why the event occurred, and that the handler might need.

Different event types will need different "supplementary information", so there are lots of classes that define the various sets of "supplementary information"; a specific event will create an object of the appropriate class to be passed to the event handler as its second parameter - and you must define the handler using the right type for that second parameter.


event occurs in the real world
Tradestation platform program
detect
route to appropriate object
object
object
object
object
call nominated method
create
object holding supplementary info about event
EL program
event handler method
sender
args
return

Unfortunately, presumably to make the platform code simpler, the first parameter to an event handler is always a generic **Object**, not a specific type of object; if a **ComboBox** sends an event notification for a **SelectedIndexChanged** event, the event handler for that event may well want to access the sender control to find out, say, what the new selected index value is, but the first parameter is only "an object" not "a combobox object". To access properties of the sender object, you need to either make extensive use of astype, eg

```
if ( sender astype comboBox ).selectedIndex = 3 then ....
```

or (my preference if there is a need to access the sender object more than once) declare a local variable of the appropriate type in the event handler and assign the sender to the local variable

```
Method void combo_SelectedIndexChanged( Object sender, EventArgs args )
vars:
ComboBox source;
begin
source = sender astype ComboBox;
if source.SelectedIndex = 3 then .....
```

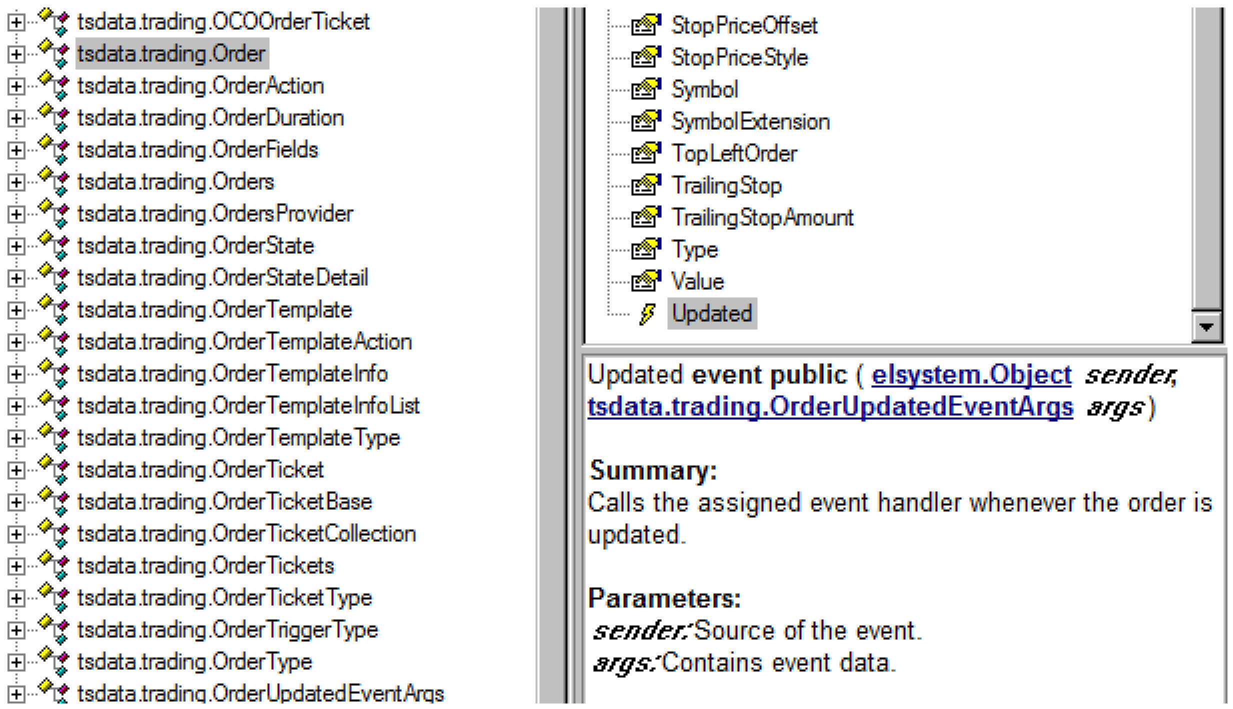
Note that it is an object within the TS platform program that calls the method in the EL program, not the EL program; the method has access to the EL program's variables, and can call other methods, but when execution reaches a **return** statement or the final **end** in the event handler method, control returns to the TS platform program. The event handler method is executed independent of any execution path through the program as a whole.

Execution of the event handler may occur at any time. One feature of normal EL is that, when the EL Program's code is evaluated as a result of a price tick, main program variables are set back to the value they had at the close of the previous bar - only at the current bar's close are the results of EL execution preserved in variables to be referenced from future bars' code. If an event handler method modifies objects within the EL program, those changes are preserved because objects are exempt from the rollback process variables experience, but if an event handler method (or any method that is called from the event handler before control returns to the TS platform program) modifies the value of a main program variable, that change will be lost at the return unless it is declared **intrabarpersist**.

> As "execution of an event handler may occur at any time", what happens if an event occurs while a different event is still being processed (ie control has not yet returned from an event handler to the TS platform)? We're told that event notifications are queued, the arrival of a new event notification can't interrupt the processing of an earlier event, the new one has to wait until the "in progress" one completes. I'm not convinced that this is universally true, although I've never located a problem caused by an interruption; it seems to me that for the ....Provider classes, a state change event interrupts an in-progress update event, for instance.

Programming and mistake location are more difficult in "event driven" programs than conventional EL programming, because the sequence of activity is no longer determined just by the code; sometimes, A can happen before B, sometimes B can happen before A.

Objects that have events show that by a lightning-bolt icon in the TDE Dictionary. Eg for an Order object



tsdata.trading.OCOOrderTicket
tsdata.trading.Order
tsdata.trading.OrderAction
tsdata.trading.OrderDuration
tsdata.trading.OrderFields
tsdata.trading.Orders
tsdata.trading.OrdersProvider
tsdata.trading.OrderState
tsdata.trading.OrderStateDetail
tsdata.trading.OrderTemplate
tsdata.trading.OrderTemplateAction
tsdata.trading.OrderTemplateInfo
tsdata.trading.OrderTemplateInfoList
tsdata.trading.OrderTemplateType
tsdata.trading.OrderTicket
tsdata.trading.OrderTicketBase
tsdata.trading.OrderTicketCollection
tsdata.trading.OrderTickets
tsdata.trading.OrderTicketType
tsdata.trading.OrderTriggerType
tsdata.trading.OrderType
tsdata.trading.OrderUpdatedEventArgs

StopPriceOffset
StopPriceStyle
Symbol
SymbolExtension
TopLeftOrder
TrailingStop
TrailingStopAmount
Type
Value
Updated

Updated **event public** (elsystem.Object *sender*, tsdata.trading.OrderUpdatedEventArgs *args*)

Summary:
Calls the assigned event handler whenever the order is updated.

Parameters:
sender: Source of the event.
args: Contains event data.

The summary shows the type of event supplementary object that will be received by an event handler when it is called. All objects of the class will be notified if an event occurs for them, but whether the EL program is told about the event is optional. Each object that can experience events has a property which Microsoft calls a "delegate", which is like a **TokenList** but for method names rather than "words". Delegates have unique syntax. For the order class shown above, to connect a specific order object to a specific method in the EL program you'd need to execute

```
myOrder.Updated += myHandler;
```

Here, **myHandler** is the name of the method (which must be declared with the correct parameter types) - **NOT** followed by ( ), because the method is not being **called** by this statement - and the **+=** operator adds its name to the end of the list of methods held in the objects **Updated** delegate (there is also a **-=** operator for removing a method name from the delegate list, but I haven't found a way to discover what is in the list as a whole). When the object is notified that an Updated event has occurred, the object calls all the methods in its Updated delegate; in normal programming the programmer will have only added one method to the delegate, but it is technically possible to add more than one and all will be called; I haven't thought of a use for that, yet.

When coding event handlers for clicks on form controls, its not unreasonable to write a unique event handler for each control. Eg (with suitable declarations)

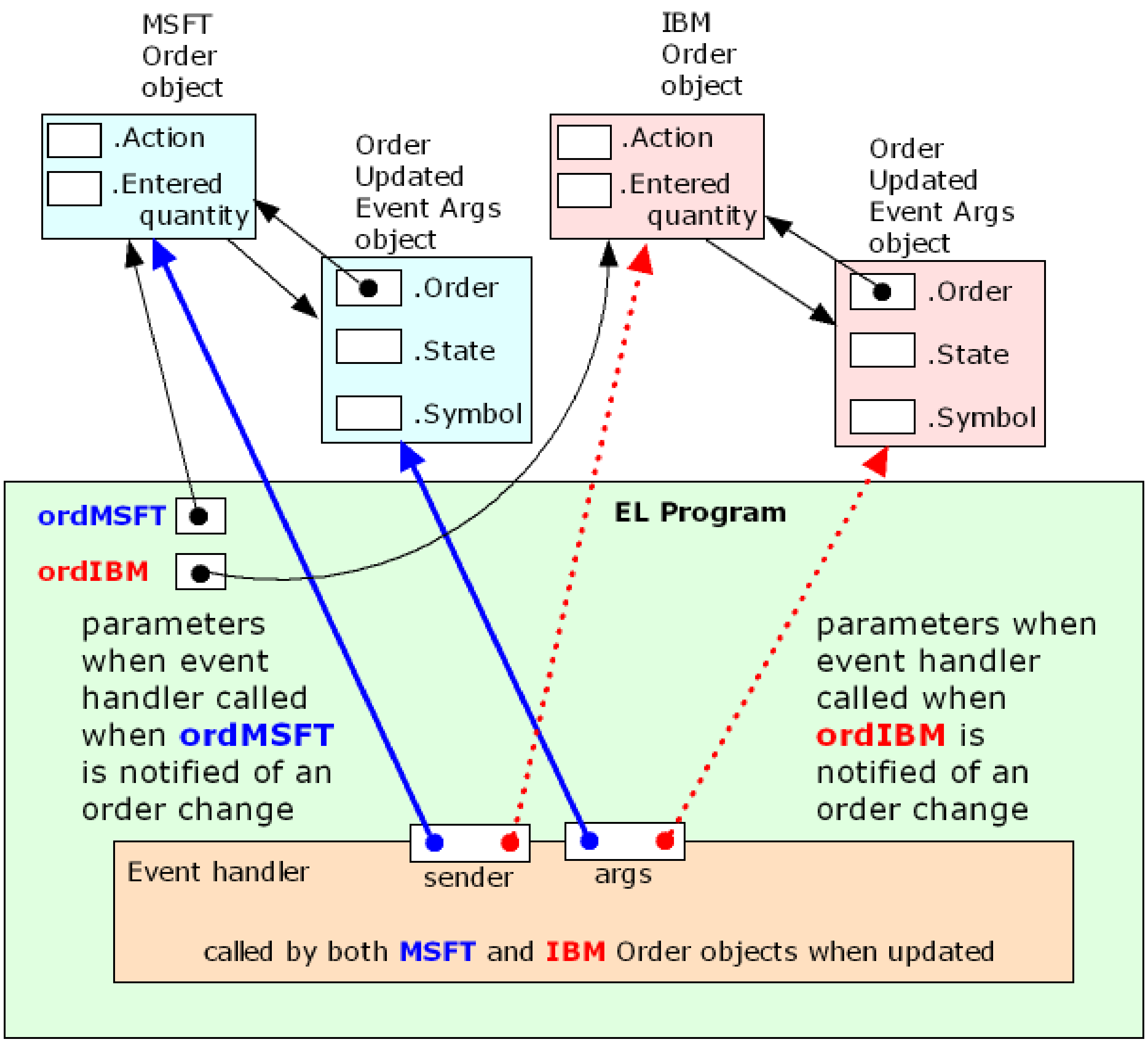
```
Method void click1( Object sender, EventArgs args )
begin
```

```
print( "click1" );
end;

Method void click2( Object sender, EventArgs args )
begin
print( "-------------- click2" );
end;
...
Once Begin
frm = Form.Create( "shared handler", 200, 200 );
frm.show( );
b1 = Button.Create( "Click", 50, 20 );
b1.click += click1;                          // tell Button b1 to call method click1 when it is clicked
frm.AddControl( b1 );
b2 = Button.Create( "Click too", 50, 20 );
b2.click += click2;                          // tell Button b2 to call method click2 when it is clicked
frm.AddControl( b2 );
end;
```

Of course, in a real program, the click handlers would do something a bit more useful. (The adventurous reader might try adding "b**1**.click += click**2**;" to the program's **once** block, to see that then clicking the first button calls both event handlers).

For classes like **Order** and **Position**, it would be difficult to code a unique handler for each **Order** and **Position** that will be created, in advance. If you wanted, for instance, to report when an order was filled, it is possible to hook multiple objects up to a single event handler, and use the **sender** first parameter of the event handler to get information about the specific order object, out of the tens that may exist, that has just filled.


MSFT Order object
.Action
.Entered quantity
Order Updated Event Args object
.Order
.State
.Symbol
IBM Order object
.Action
.Entered quantity
Order Updated Event Args object
.Order
.State
.Symbol
ordMSFT
ordIBM
EL Program
parameters when event handler called when ordMSFT is notified of an order change
parameters when event handler called when ordIBM is notified of an order change
Event handler
sender
args
called by both MSFT and IBM Order objects when updated

```
Method void orderUpdate( Object sender, OrderUpdatedEventArgs args )
vars:
Order source;
begin
source = sender astype Order;
If args.State = OrderState.rejected then begin
print( args,Symbol, " ", source.Action.ToString( ), " order rejected ***********" );
return;
end;
if args.State = OrderState.filled then
print( args,Symbol, " ", source.Action.ToString( ), "order of ", source.EnteredQuantity:0:0, "
filled" )
else if args.State = OrderState.partiallyFilled then
```

```
print( args,Symbol, " ", source.Action.ToString( ), "order partially filled to ",
source.FilledQuantity:0:0 );
end;
....
ordMSFT = ticketMSFT.Send( );
ordMSFT.Updated += orderUpdate;
ordIBM = ticketIBM.Send( );
ordIBM.Updated += orderUpdate;
```

should be capable of producing PrintLog output like

```
IBM Buy order rejected ***********
MSFT Buy order of 50 filled
```

Earlier, I described **Const** for defining a named constant value. The example above uses an **enumeration** - a set or related constant values grouped together under one name by using the same dot notation as is used for accessing items within an object. The **OrderState** enumeration defines a number of different words that are appropriate for the different values that an order object's **State** property can take on; OrderState.Rejected, OrderState.Filled etc. It is good practice to always use enumeration names when an enumeration is defined.

Enumerations are not classes, you can't use them to create objects, but you can declare variables using them. Unfortunately, enumerations are not yet fully integrated into the language, they're still regarded as fancy names for **int** values. For instance, "Help" tells you that Orderstate.partiallyFilled = 6, and

```
OrderState myState( OrderState.partiallyfilled ),
OrderState myState( 6 ),
```

are equally valid declarations. I think this leads programmers into the temptation of using the **int** numbers to save typing, which makes mistakes less obvious as described for **Const**; trying to test for **partiallyFilled** by

```
if args.State = 7 then ...
```

makes it harder to spot that this is wrong (because 7 is actually OrderState.partiallyFilledURout). Fewer mistakes would be made if strong typing was applied to enumerations, so the above **had** to be written

```
if args.State = OrderState.partiallyFilled then ....
```

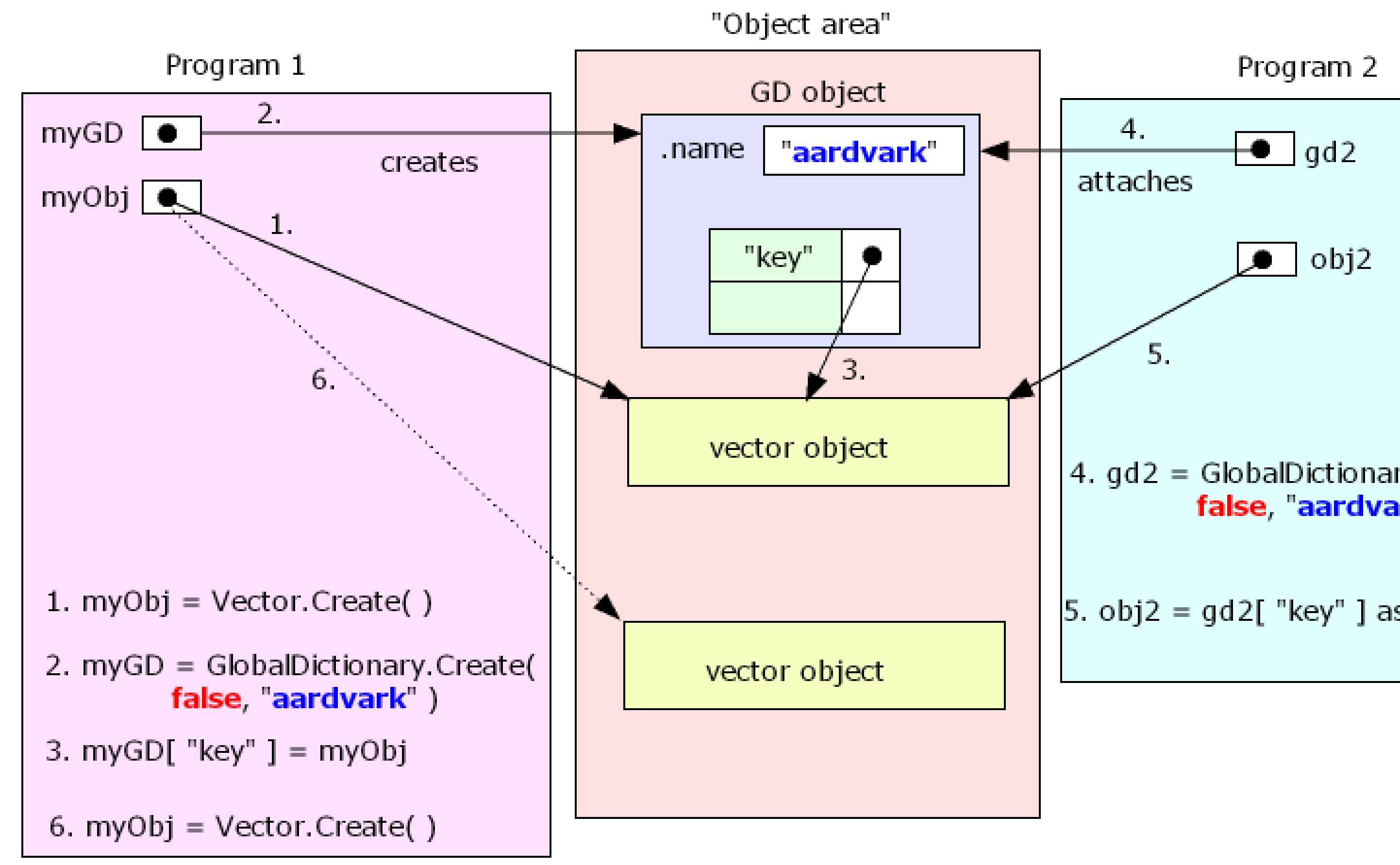
And currently, there are places where OrderState.Filled wouldn't be acceptable syntax (eg saving it in AppStorage), you have to use the **int** values, and the **astype** keyword isn't valid on enumerations currently; you have to retrieve the value by myState = AppStorage["myState"] astype int;

because

myState = AppStorage["myState"] **astype OrderState**;

won't verify. I hope the language will be updated in the future, but I fear it won't because TS won't want to break code written with the laxer standards when they tighten the standards. Lets just hope they don't change the int values in any enumerations, because that will break existing code, too.

A **GlobalDictionary** can be seen as a way for one EL program to publish values that can be read by other EL programs, as the various "global variable" DLLs or the sparsely-documented **GlobalValue** component do. But it also has events that can alert a program when one of these published values is changed, to provide the basis of a messaging system between EL programs. When created via **GlobalDictionary.Create**(**false**, ...) it can also contain objects (allowing collections to be stored in a GD created via **GlobalDictionary.Create**(**true**, ...) - needed to share between RadarScreen and charts, or between charts when multi-code is enabled - is beta test Update 17).



When running on a chart or radarscreen, any objects used by an EL program are held within the host "TS application" instance; if an indicator on a chart creates an object, the object is created within ORCHART's RAM allocation. Two EL programs in the same TS application then have their objects in the same place, and so in principle both programs could access the same object. The **GlobalDictionary** provides a mechanism for achieving that.

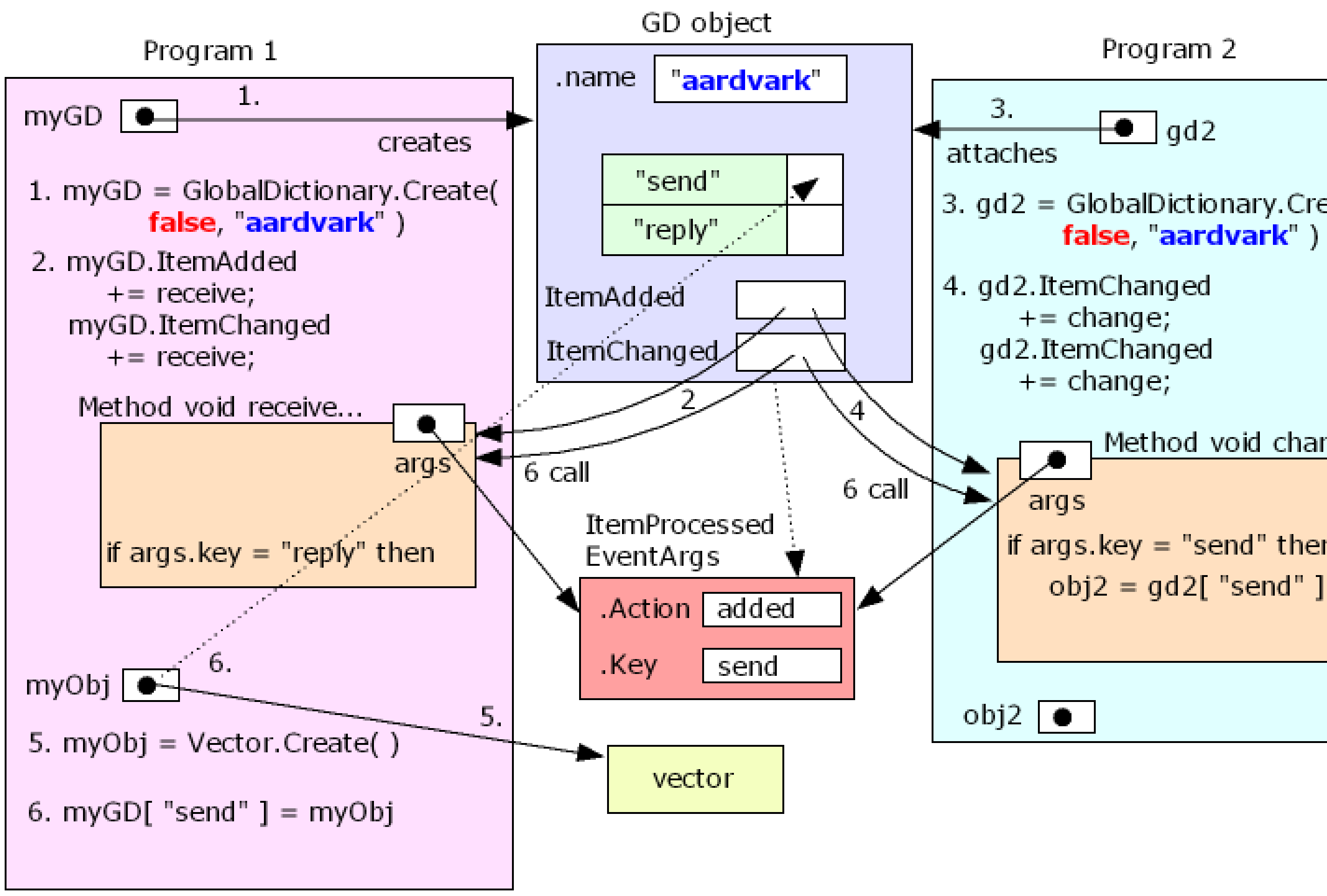
1. program 1 creates a **Vector** object; and saves the pointer to that object in a variable
2. program 1 calls **GlobalDictionary.Create**(**false**, "**aardvark**"); the **false** signals that this GD is going to be shared with other programs in the same TS application, so the GD object is to be created within the current TS application. A GD object (like several others, such as the various Provider objects) has a **name** Property, apparently for use by the TS platform as the EL programs can't see that property. When a program uses the **Create** method, it is allowed to provide a string as a parameter and the platform will check if there are any existing GD objects that have that value in their .**name** property. If none do, then the **Create( )** call will create a new GD object and put the string into the object's **name** property. This is the case in program 1, so the object is created and the pointer to the object is saved in a program 1 variable.
3. Program 1 assigns its **vector** reference to an item in the GD object. As with a normal Dictionary, a "key string" is provided to identify the item; if an item using that key string already exists, the item will be overwritten, otherwise (as here) a new item will be created. The pointer from the named variable on the right of the = will be copied into the GD item, so now there are two pointers to the vector object
4. Program2 calls the GD method **GlobalDictionary.Create**(**false**, "**aardvark**"), the same parameters that Program 1 used. This time, the check for "GD objects named **aardvark**" finds that there is one, so rather than **create** a GD object, the **Create**( ) method simply returns a pointer to the **existing** object, which then Program 2 saves in one of its variables. Both programs now have pointers they can use to access a single GD object.
5. Program 2 reads the current contents of the GD item associated with the key string "key"; those contents are "a pointer to an object". Program 2 could use the **istype** test to check what sort of object the pointer actually points at, or if it is confident there are no programming errors it could simply go ahead and promise the Verifier that the pointer will, when the program runs, be a pointer to a Vector by using the **astype** keyword as shown. This allows Program 2 to save the pointer in a variable declared as type **Vector**. There are now three pointers to a single **Vector** object - one in program 1, one in the GD, and one in program 2. Both programs can see values in the vector, by using their pointers to it.
6. Program 1 creates a second **vector**, and saves the pointer to that **vector** in the same variable it used earlier for the vector it saved in the GD. That new **vector** is currently private to Program 1, program 2 can't see it. If program 1 wrote the pointer into a new item in the GD then it would make it available to program 2. If program 1 read the "key" item out of the GD into the same **myObj** variable, or into a different variable, it could regain access to the shared **vector**.

Of course a GD item can contain int, double, bool or string values, but they don't need a diagram to illustrate the process. The GD can also hold pointers to other types of object; in RadarScreen each symbol has its own set of variables for an indicator, and you can share single instances of objects like timers or forms between symbols, rather than having each symbol create its own objects, by putting the objects in a GD they all share.

GlobalDictionaries have delegates so that the programs that share them can attach event handlers to the GD, to be called when appropriate. There are separate delegates for ItemAdded, ItemChanged and ItemDeleted. However, all three events pass an **ItemProcessedEventArgs** object to the event handler method that is called, which includes an **Action** property having possible values ItemProcessed.Added, ItemProcessed.Changed, ItemProcessed.Cleared, ItemProcessed.Deleted, and ItemProcessed.Unknown, so a single event handler method can deal with (more than) all the event types. The object includes a property that gives the name of the item that experienced the event, to help the event handler deal with it.

When a GD signals that an event has occurred, it calls **all** the event handlers that have been hooked up to that event. The simplest approach is to have one GD for messages from A to B and one for replies from B to A; the next simplest is to allow A and B to both write to the same GD, but not both write to the same item in the same GD. If A and B both hook themselves to **ItemChange** events of the same GD, and both change the same item, there is a risk that you'll set up an infinite chain of A replying to A's reply to A ..., or A replying to B which was replying to A ....

If you add a collection to a GD item, changing an item in the collection will NOT cause a GD **ItemChanged** event; the event only happens when you change the contents of the item in the GD, not something in the object that item points at.



1. Program 1 causes a GD object to be created, which allows sharing between programs in the same TS application
2. Program 1 hooks up the GD **ItemAdded** and **ItemChanged** events to call Program 1's **receive**( ) method
3. Program 2 get a pointer to the same GD object
4. Program 2 hooks the GD's **ItemAdded** and **ItemChanged** events up to call program 2's **change**( ) method; this is **in addition** to the hook ups to Program 1's methods.
5. Program 1 creates a vector
6. Program 1 copies the pointer to the vector into a new item in the GD. This causes the GD to create an **ItemProcessEventAgrs** object, set its **Action** property to **ItemProcessedAction.Added** (abbreviated in the diagram) and the **Key** property to "**send**" (the key string that identifies the item that was added); it then calls **both** the methods included in the **ItemAdded** delegate, passing a pointer to the **ItemProcessEventAgrs** object to each method (plus a pointer to the GD, which isn't shown on the cramped diagram above). The two methods check the details in the object they receive, one sees that the .**Key** property is not "**reply**" so it does nothing, the other sees that the .**Key** property is "**send**", so it responds.

For a GD created via .Create(true, ...), the GD is intended to share information between EL programs running in different TS applications (say between RadarScreen and a chart, or between charts controlled by instances of OrChart running on different cores if multi-core is enabled. EL objects created within one host applications will be created within that host application, and an EL program hosted in a different application cannot use a pointer to access objects in the first application. The GD will not allow sharing of objects between applications in the direct way described above.

Serialisation and AppStorage

Information in a file on a disk or CD is stored as a long sequential stream of bits (0s and 1s). Breaking the stream up into groups of 8 makes it look like a stream of bytes. If the bytes are intended to represent text, with one character per byte, the file would be a single very long line; however, by giving one possible byte value the meaning "start a new line" that one line can be broken up into multiple lines, so the file can be seen as a column of shorter lines of text.

A web page, such as this one, is defined by text that conforms to the rules of the HTML "language". You can see the text by opening this file with Notepad. HTML is a combination of text to be displayed, and control "tags" embedded in the stream of text; when a browser is displaying the web page, when it recognises a tag it starts displaying the text that follows the tag in a way dictated by the meaning of the tag, until it gets to a "closing tag" that marks the end of the effect of the tag.

Tags are represented by a word enclosed in angle brackets, < and >. The first word in the tag can be selected from a range of predefined words, each of which has a layout meaning that the browser will apply when displaying the HTML. The closing tag, that marks the end of the display effect, is also enclosed in angle brackets and the first word is the same as the word in the opening tag it matches, but with a **/** character in front. Thus, in HTML <b>aardvark</b> will cause a browser to display the word aardvark bolded as **aardvark**. A tag that isn't intended to "enclose a block" can combine the "opening tag" and "closing tag" functions into one tag by ending with a /> rather than just >. There are dozens of pre-defined tag names in HTML, allowing the simple stream of text that comes through the internet to your browser to describe all the 2-d visual effects that you see in your web browser.

As well as marking the start of a block of text to be displayed in a specific way, a tag can also have "attributes" stored inside it; predefined words that can be included within the < and > of the tag, followed by an = character and a value enclosed in double quotes. Eg <div **style=**"indent"> marks the start of a block of text whose appearance will be controlled by a "style rule" named indent specified by an attribute.

XML follows the ideas of HTML, but without all the "predefined meanings". XML has opening and closing tags (words within < and >) to identify a block of content, with optional named attributes within the tag, but the author has complete freedom to select tag names and attribute names without restriction (apart from not including spaces or punctuation marks, that would interfere with them being recognised as "words"). The aim of XML is to provide a way of describing structured information, in which there can be containers within containers within ...., using simple text that can be stored in a disk file or transmitted over a network. Because the user can make up their own words, there is no limit to the type of data that can be described, and if the words are well-chosen then the description should be relatively self-documenting.

Classes define containers that contain properties, and some properties my contain further properties. Objects created from the classes can use pointers to "include" (a reference to) one object from another. The pointers themselves are only significant while the objects are in RAM. To describe data structures built from objects and references to objects, in a form that can be stored on disk or transmitted from one computer to another over a network, requires the structure to be expressed in a different way.

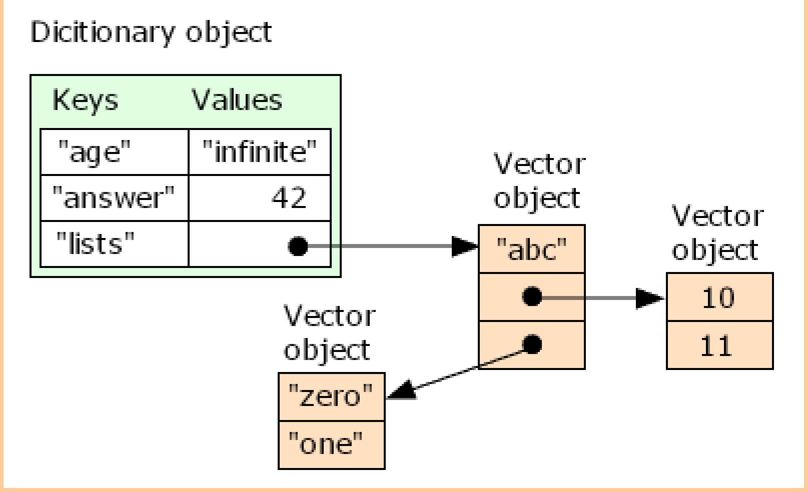
For instance, suppose you had a Form object that contained a Button object. You could describe (an abbreviated version of) that structure by a text string

```
<Form title="serialised">

  <size w="300" h="200" />
  <location x="42" y="54" />
  <BackColor r="128" g="128" b="128" />
  <Button text="Click me">
    <size w="60" h="20" />
    <location x="0" y="0" />
    <BackColor r="255" g="0" b="0" />
    <ForeColor r="255" g="255" b="255" />
    </Button>
  </Form>
```

which could be stored in a disk file (as the above has, so that you can read it), and a suitably-written program could read the text description and use it as a recipe to recreate the original form and button.

The same process could be applied to structured data such as collections. For instance, given a data structure



this could be described in a string written to a file, as

```
<Dictionary count="3">
  <item key="age">
    <string value="infinite" />
    </item>
  <item key="answer">
    <int value="42" />
    </item>
  <item key="lists">
    <Vector count="3">
      <string value="abc" />
      <Vector count="2">
        <int value="10" />
        <int value="11" />
        </Vector>
      <Vector count="2">
        <string value="zero" />
        <string value="one" />
        </Vector>
      </Vector>
    </item>
  </Dictionary>
```

and again, a program could be written that would read through this string and reconstruct the data structure, converting "containment in the description" into "pointer to a created object" along the lines of

```
Method Vector newVector( )
begin
create new vector
repeat
```

```
get next tag
switch tag
Case "int": value = numtostr( extracted value string, 0 ); vector.push_back( value );
Case"string" : value = extracted value string; vector.push_back( value );
Case"Vector" : value = newVector( ); vector.push_back( value );
end;
until tag = "/Vector";
end;

if tag = "Dictionary" then begin
create a dictionary object
repeat
get next tag
extract key string
switch tag
Case"int" : value = numtostr( extracted value string, 0 ); dictionary[ key ] = value;
Case"string" : value = extracted value string; dictionary[ key ] = value;
Case"Vector" : value =newVector( ); dictionary[ key ] = value;
end;
until tag = "/Dictionary"
end;
```

AppStorage is a pre-declared Dictionary, with the extra property that will be automatically preserved "when an analysis technique is closed-reopened, re-verified, or has its inputs modified", according to "Help". It appears to do the same across a data disconnect, and a change of chart symbol, although they are not mentioned in "Help". It is private to the analysis technique, not a sharing mechanism like the GlobalDictionary. Not explicitly stated, but behind the "name-value pairs" it will preserve at least some objects, presumably by serialising them.

I've been able to save a Form object in AppStorage (which I think solved a problem trader5615 was having, whereby he was crashing TS by re-creating his form each time he changed symbol, and hitting a 10,000 max Windows User Objects limit because the form controls were not being released when a new copy of the form was created), although I had to re-assign event handlers to form controls on restoration, and save Vectors and Dictionaries.

```
Method Vector restoreVector( string key )
begin
If Appstorage.contains( key ) = false then return new Vector;
if Appstorage[ key ] = Null then return new Vector;
return Appstorage[ key ] astype Vector;
end;

Method Dictionary restoreDictionary( string key )
begin
```

```
If Appstorage.contains( key ) = false then return new Dictionary;
if Appstorage[ key ] = Null then return new Dictionary;
return Appstorage[ key ] astype Dictionary;
end;

Method TokenList restoreTokenList( string key )
begin
If Appstorage.contains( key ) = false then return new TokenList;
if Appstorage[ key ] = Null then return new TokenList;
return Appstorage[ key ] astype TokenList;
end;

method void AnalysisTechnique_Initialized( Object sender, InitializedEventArgs args )
begin
chartMessages = restoreDictionary( "chartMessages" );
unsentOrderLetters = restoreVector( "unsentOrderLetters" );  // chart letters that requested
the unsent order
sentOrderLetters = restoreDictionary( "sentOrderLetters" );    // chart letters that triggered the
order send, keyed by orderID
reportableOrders = restoreTokenList( "reportableOrders" );   // list of sent orderIDs, generated
by the aggregator
end;

method void AnalysisTechnique_UnInitialized( Object sender, UnInitializedEventArgs args )
begin
AppStorage[ "chartMessages" ] = chartMessages;
AppStorage[ "unsentOrderLetters" ] = unsentOrderLetters;
AppStorage[ "sentOrderLetters" ] = sentOrderLetters;
AppStorage[ "reportableOrders" ] = reportableOrders;
end;
```

So far I haven't tried to catalog what types of object can and can't be saved in **AppStorage**, all the **Vectors** and **Dictionaries** hold strings to be safe. It was necessary to check more than just .**Contains** to successfully restore a collection (hence the restore....( ) methods above).

Update 17 seems to be introducing **AppStorage**-style serialisation for storing at least collections in a **GlobalDictionary** created by .**Create**(**true**, ...).

OrderTickets, Orders, Positions and Providers

The **OrderTicket** class represents a message you would like to send to a TS server in order to create an order. It has a **Create** method. An **Order** object represents a receiver for messages from a TS server, and doesn't have a **Create** method - the only way to get access to an **Order** object is to receive a pointer to the object from the TS platform. To create an order at the server, you create an **OrderTicket** object in the local PC, and call its **Send**( ) method; this will transmit the ticket to the servers, create an **Order** object in the PC to receive status updates from the server, and return a pointer to that **Order** object as a result.

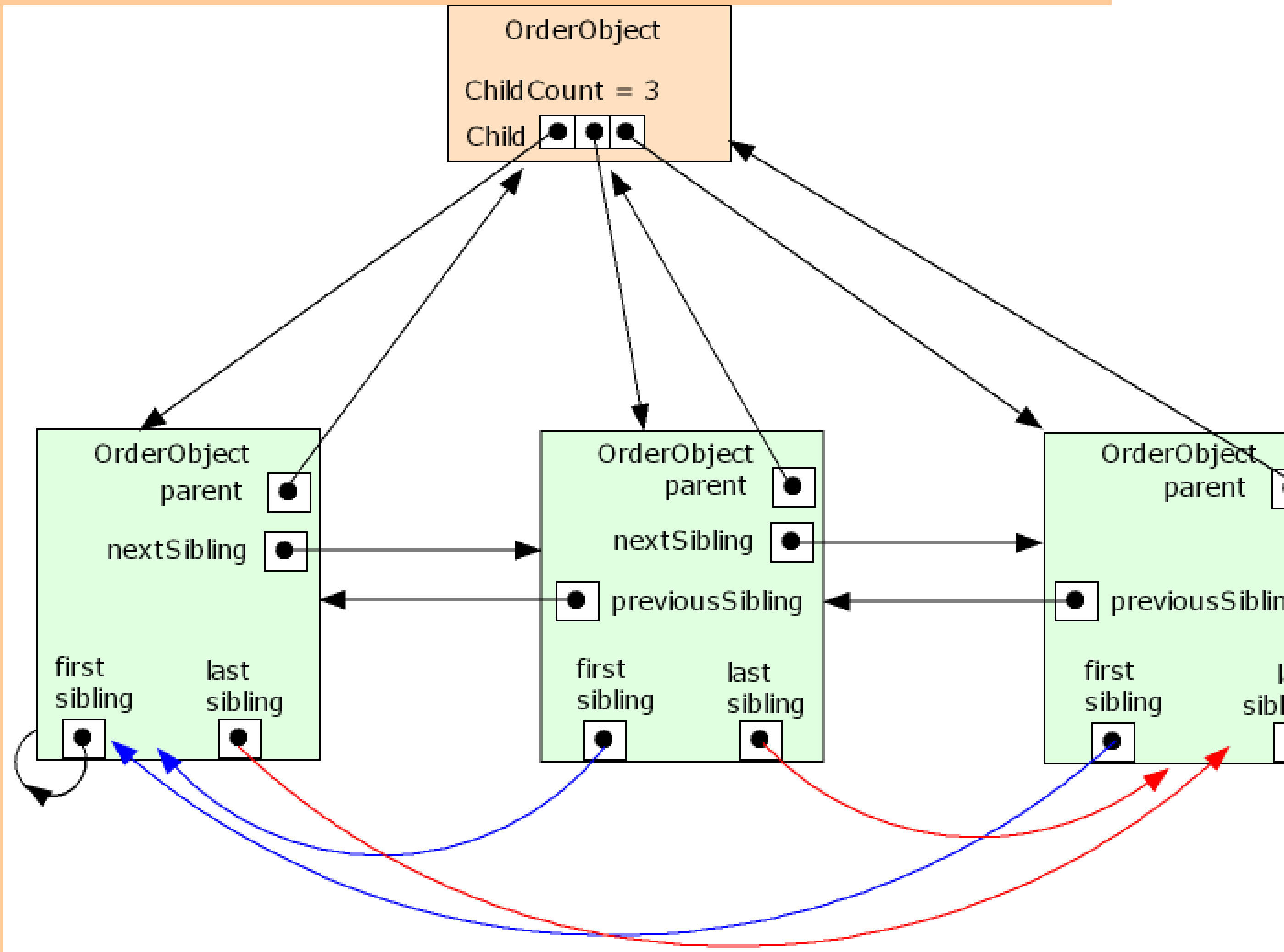
An **OrderTicket** may actually create more than one **Order** object, as a related group. The programming terminology relies on words from family trees, so an order that creates several orders sees them as "children", the original order is the "parent", and the children are regarded as "siblings".

An **Order** object has a property **ChildCount** that, for an OSO/OCO order, tells you how many child orders it has; the individual child orders can be located using the **Child** indexer by giving an int number in [ ]. An **Order** object has a **Parent** property, that will be blank for the order that actually is the **Parent**, but in the children will point at their parent. There are also properties **FirstSibling**, **LastSibling**, **NextSibling** and **PreviousSibling** that are set to organise multiple children of a parent into some sort of order. The limit of my sophistication is an OSO order that contains a market entry order with a bracketed stopLoss/Target pair, so I haven't had to use the sibling links. For logging the "OSO market entry + bracket exits", I create a dictionary representing a trade :

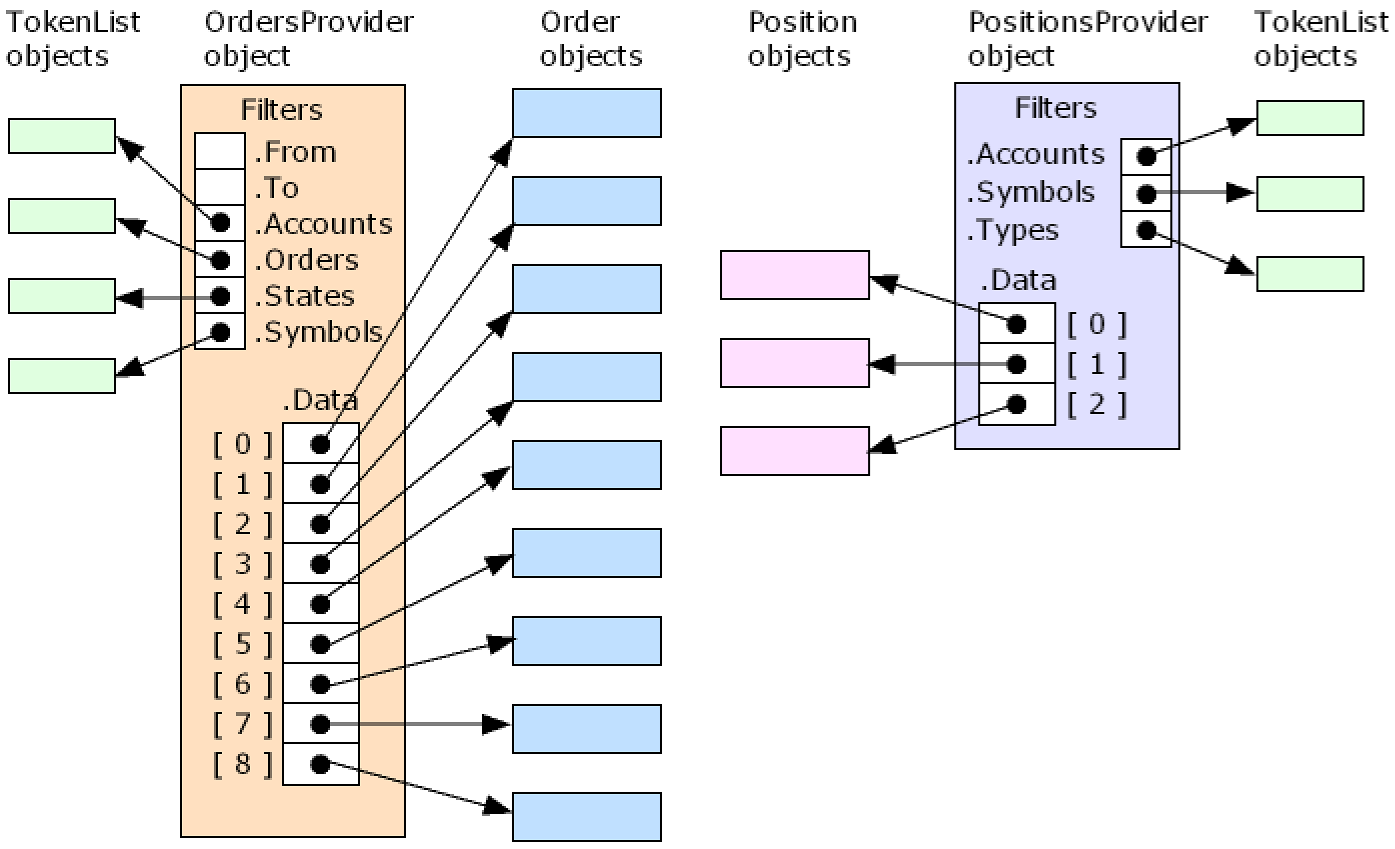
```
Method Dictionary newTradeObject( Order entryOrder )
vars:
int index,
Dictionary trade,
Order stopOrder;
begin
trade = New Dictionary;
trade[ "entry" ] = entryOrder;
For index = 0 to entryOrder.childCount-1 begin                          // link child orders to
shared order Updated event handler
entryOrder.Child[ index ].Updated += orderUpdate;
If entryOrder.Child[ index ].Type = OrderType.limit then
trade[ "target" ] = entryOrder.Child[ index ]              // save profit target order in the trade
"pseudoObject"
Else begin
stopOrder = entryOrder.Child[ index ];
trade[ "stop" ] = stopOrder;                               // save stopLoss order in the trade
"pseudoObject"
If orderDirection( entryOrder ) = OrderAction.Buy then
lastLongStop = stopOrder.StopPrice                 // save initial stop ready for trailing
Else
lastShortStop = stopOrder.StopPrice;
If lastShortStop = 0.0 then lastShortStop = 99999.99;     // just to be safe
end;
```

```
end;
Return trade;
end;
```

If I knew how to create an order with three children I'd expect it to look like

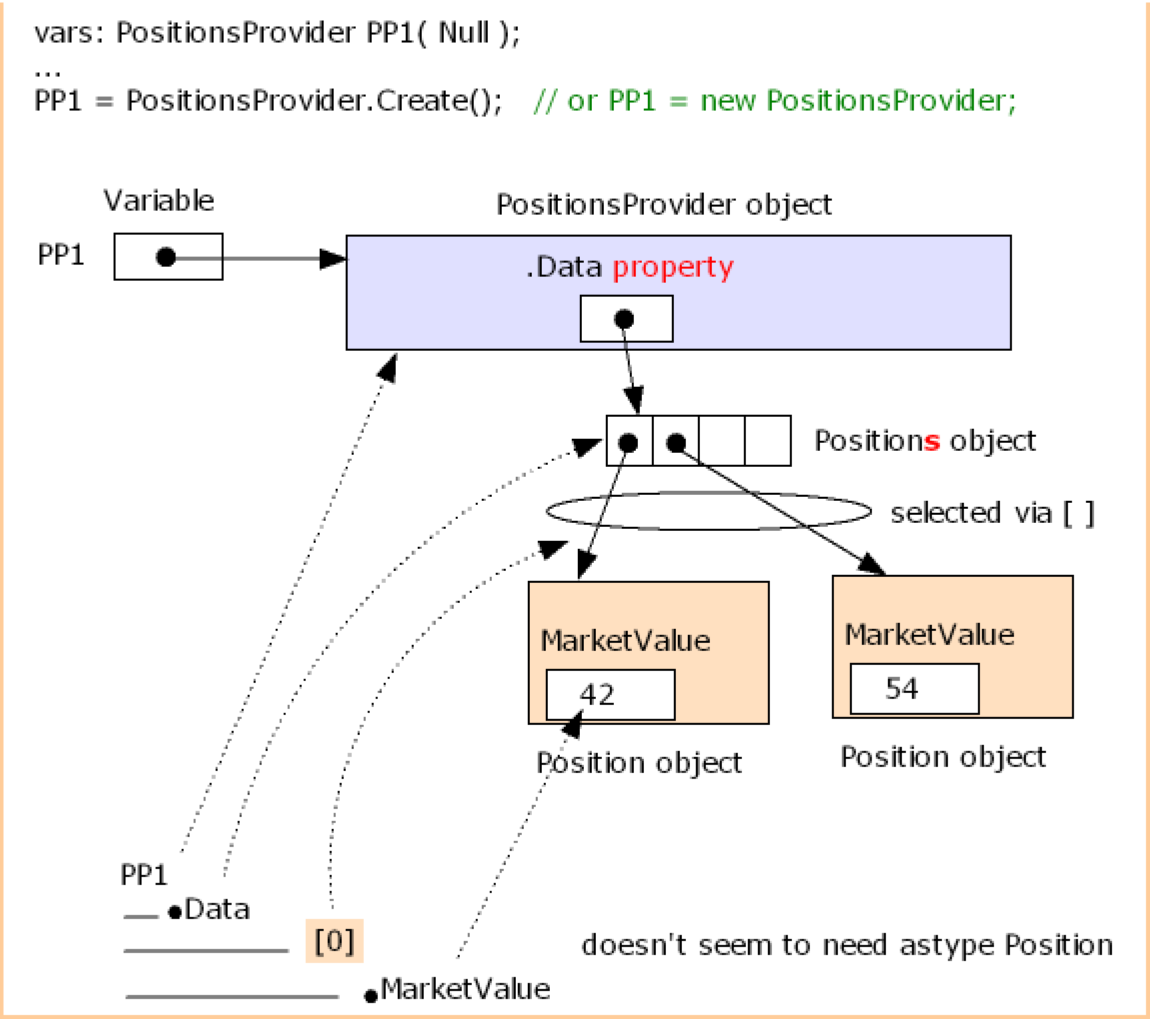


No doubt the names "OrdersProvider" and "PositionsProvider" have a logic within the TS development department, but they seem more like indices to all the available **Order** and **Position** objects (or a sub-set, if you use the various filtering properties). Both provide indexers (search methods for a collection that accept a parameter in [ ] rather than ( ), and return a result of a specific type of object) to locate a single object within the index. Actual information is held in the **Order** and **Position** objects.



There are classes **Orders** and **Positions** defined, representing collections based on **Vector** but contrived to return **Order** or **Position** objects rather than generic **Object** objects; they have no **Create**( ) method so you can't create your own objects from them. I assume they were created to implement part of the functionality of the **OrdersProvider** and **PositionsProvider** classes, but I can't see that their "Help" entries add any value. They just mean you have a choice of ways of accessing **Order** and **Position** objects. Using the **PositionProvider** as an example, but **OrdersProvider** implements the same principle :

Access using a hierarchy of object references to "things inside things inside things ..." :

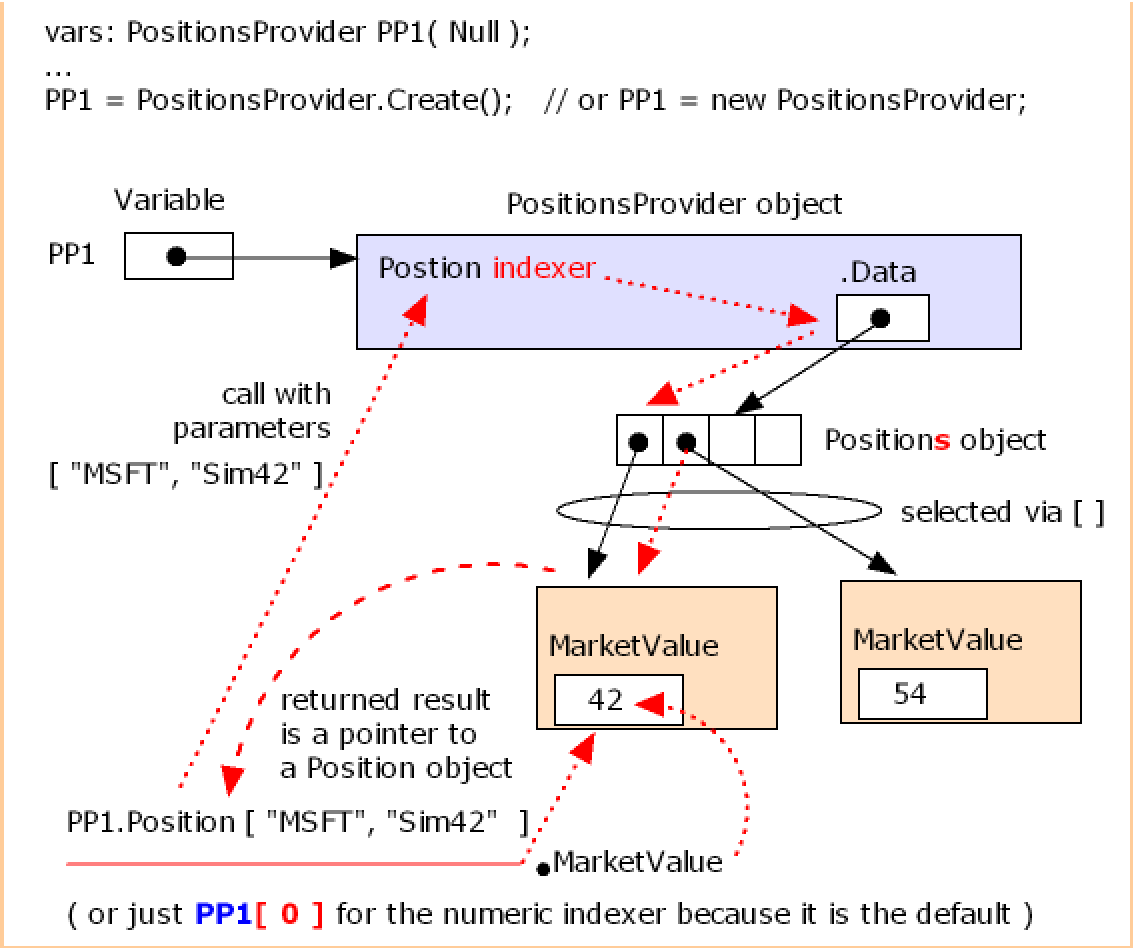


```
vars: PositionsProvider PP1( Null );
...
PP1 = PositionsProvider.Create();   // or PP1 = new PositionsProvider;
```

"PP1.Data[0].MarketValue" is executed by

1. go to variable **PP1** and follow its pointer to an object
2. access the **Data** property in that object and follow its pointer to a variation of a vector
3. apply the [0] to select one pointer from the vector, and follow it to a **Position** object
4. access the **MarketValue** property inside that object

Access using an indexer :



```
vars: PositionsProvider PP1( Null );
...
PP1 = PositionsProvider.Create();   // or PP1 = new PositionsProvider;
```

Variable
PositionsProvider object
PP1
Postion indexer
.Data
call with parameters ["MSFT", "Sim42"]
Positions object
selected via [ ]
MarketValue
42
MarketValue
54
returned result is a pointer to a Position object
PP1.Position ["MSFT", "Sim42"]
.MarketValue
(or just PP1[0] for the numeric indexer because it is the default)

"PP1.Position["MSFT", "Sim42"]" is executed as

1. go to variable **PP1** and follow its pointer to an object
2. call the **Position** indexer method, passing "MSFT" and "Sim42" as parameters
3. the indexer follows the pointer from the **Data** property to the vector-like **Positions** collection object
4. the indexer searches through the items for one that matches the indexer parameters in its .**Symbol** and .**AccountID** properties
5. a pointer to the matched object is returned as a result, to be used in place of "PP1.Position["MSFT", "Sim42"]"
6. the returned pointer is followed to the object it points at, and the .**MarketValue** property inside that object is accessed

A similar principle would apply to the numeric indexer **PP1.Position[0]** - or just **PP1[0]** as the .Position part can be omitted to use the default indexer- except that it probably doesn't have to "search", it can just go straight to the indexed item in the Vector-like **Positions** collection.

DataProvider interface in **tsdata.Common**, inherited by **OrdersProvider** and **PositionsProvider**, among others

bool Load	change value to cause the provider to load data from, or disconnect from, servers
bool Realtime	true if data provided by the provider is real-time, false if not
DataState State	the state of the provider (DataState.failed, DataState.loaded, DataState.loading, DataState.unloaded)
TimeZone TimeZone	the time zone of the provider, TimeZone.Exchange or TimeZone.Local
StateChanged	delegate allowing the provider to be connected to a method to be called when its state property changes value

OrdersProvider interface :

TokenList Accounts	accesses a pointer to the **TokenList** object that contains a list of **accountID**s to be used to filter the provider contents
bool Contains(order)	true if the **OrdersProvider** contains a reference to that **Order** object
bool Contains(string)	true if the **OrdersProvider** contains a reference to **Order** object with that **OrderID** string
int Count	number of **Order** objects the **OrdersProvider** indexes
OrdersProvider Create( )	creates an **OrdersProvider** object and returns a pointer to that object
Orders Data	unclear; "Help" says its a reference to a class **Orders** collection of **Order** objects, presumably for allowing access without using the **Order** indexers, but not sure why; includes its own **Contains**( ), **Order**[ ] and **TryOrder**( ) methods? "Data" to avoid the name clash with "Orders"?
DateTime From **DateTime** To	**DateTime** objects (not a legacy **double** DateTime value) that specify the start date and time, and end date and time, of the time range for which orders are wanted; must either both have values, or neither have values (in which case they represent "all of today")
Order Order[**int**] **Order** Order[**string**]	indexers providing access to a specific **Order** object within the **Data** collection. based on a numeric position index, or matching an order whose **OrderID** property matches the string parameter. The numeric version is the default, allowing you to put [42] after an **OrdersProvider** variable name to access the order at position 42 in the index.

TokenList Orders	accesses a pointer to the **TokenList** object that contains a list of **orderID**s to be used to filter the provider contents
TokenList States	accesses a pointer to the **TokenList** object that contains a list of states to be used to filter the provider contents; being a **TokenList**, presumably these states have to be expressed as strings, not **OrderState.Filled** format?
TokenList Symbols	accesses a pointer to the **TokenList** object that contains a list of symbols to be used to filter the provider contents
Order TryOrder(string)	like **Contains**(string), searches for an **Order** object that has the specified **OrderID**, but returns either a pointer to the **Order** object - or **null** if not found. A similar feature would be useful for Dictionaries.
Updated	delegate allowing the provider to be connected to an event handler method that will be called when one of the contained orders experiences an Updated event. Use the .**Order** property of the **args** parameter to get access to the actual **Order** object that experienced the update, as the **sender** parameter presumably identifies the **OrdersProvider** as the .**Reason** values related to the **OrdersProvider**.

PositionsProvider interface, pretty much "as OrdersProvider, but for Positions"

TokenList Accounts	accesses a pointer to the **TokenList** object that contains a list of **accountID**s to be used to filter the provider contents
bool Contains(string, string)	true if the **PositionsProvider** contains a reference **to** a Position object for a **symbol** matching the first parameter and an **accountID** matching the second parameter
int Count	number of **Position** objects the **PositionsProvider** indexes
PositionsProvider Create( )	creates an **PositionsProvider** object and returns a pointer to that object
Positions Data	unclear; "Help" says its a reference to a class **Positions** collection of **Position** objects, presumably for allowing access without using the **Position** indexers, but not sure why; includes its own **Contains**( ), **Position**[ ] and **TryPosition**( ) methods?
TokenList Symbols	accesses a pointer to the **TokenList** object that contains a list of symbols to be used to filter the provider contents
Position TryOrder(string, string)	like **Contains**(string), searches for a **Position** object that has the specified **symbol** and **accountID** values, but returns either a pointer to the **Position** object, or **null** if not found.

TokenList Types	accesses a pointer to the **TokenList** object that contains a list of position types to be used to filter the provider contents; being a **TokenList**, presumably these states have to be expressed as strings, not **PositionType.LongPosition** format?
Updated	delegate allowing the provider to be connected to an event handler method that will be called when one of the contained positions experiences an Updated event. Use the .**Position** property of the args parameter to get access to the actual **Position** object that experienced the update, as the **sender** parameter presumably identifies the PositionsProvider as the .**Reason** values related to the **PositionsProvider**?