

SD125181

Advanced AutoLISP—from Hack to Expert

Darren Young
Hermanson Company

Learning Objectives

- Learn how to use advanced AutoLISP functions
- Understand advanced concepts like recursion
- Understand how reactors work
- Learn how to use the nearly 2000 undocumented AutoLISP functions

Description

Are you a casual LISP programmer? Do you get what you want done but feel like you have to bludgeon your way through it? Do you have trouble getting what's under the hood into overdrive? Then, this class is for you. You'll learn many of the things that real power AutoLISP programmers know.

Speaker(s)

Based in Washington state, Darren Young has been a veteran Autodesk University speaker for well over a decade. His unique ability to leverage multiple every day technologies in interesting ways to solve implicated and laborious tasks has been valued by users around around the world. Down to earth and approachable, he's always willing to help his peers anytime of the year even outside of Autodesk University. Darren's background includes a wide variety of disciplines such as Construction, Engineering, Manufacturing, LEAN, Information Technology, Computer Programming, Author and Technical Editor. His lectures and labs are not just a training opportunity for others but a venue which connects him personally with users helping him learn as well.



Introduction

This course is for anyone who's casually programmed AutoLISP but knows there's more untapped power under the hood. At the end of this class, you'll know exactly where to look and how to tap into that power increasing your coding abilities and enabling you to write more functional programs, with less code.

With ARX, VBA, and now .Net (Dot Net), there's always a rumor flying around that AutoLISP is dead or going away. That's simply not true. While there are plenty of reasons to use other APIs and languages, there's also plenty of reasons to continue using AutoLISP and Visual LISP. For the non-professional programmer, it's a great, easy to learn language with plenty of power. It's built into AutoCAD, it can do most of what VBA can do, and you don't have to purchase any special development software. If you want to automate AutoCAD, AutoLISP and Visual LISP is a perfectly suitable option. Take it for a test drive and put the pedal to the floor. You'll be amazed at what it can do.

Advanced AutoLISP

(apply 'function list)

This function takes a quoted function, and applies it to a list of arguments. This can be very useful for functions that require a list of arguments when you don't know how many there will be. If you know how many arguments or variables, it's typically not needed.

(+ 1 3) ; *We know there's 2 numbers to add. It's all that's needed*

(apply '+ '(1 3)) ; *This isn't very readable, efficient to code or helpful*

(setq mylist '(list 1 3)) ; *What if our data was in a list?*

(+ (nth 0 mylist) (nth 1 mylist)) ; *Now this isn't very efficient or readable*

(apply '+ mylist) ; *In this case, this is now better*

(setq mylist '(1 3 5...etc...)) ; *What if we didn't know how many items in the list?*

(apply '+ mylist) ; *It's still the same no matter how many items*

When you have a list of raw data, it's not practical to build a new list inserting the function in the front of it so that it can be evaluated. It's also not efficient (coding or processing) to loop through that list of potentially unknown length processing each item one at a time.

This...

(1 3 5)

Turned into this...

(+ 1 3 5)

Would require this...

(eval (cons (quote +) mylist))

Or require this...

(setq number 0)

(foreach item mylist

(setq number (+ number item))

)

But is simpler with this...

```
(apply '+ mylist)
```

```
(mapcar 'function list {list} ...etc...)
```

This function is similar to *(apply)* but is different in one key way, while *(apply)* applied a function to a whole list of arguments at a time, *(mapcar)* also takes a list (or lists), but applies the function to each item in the list individually, then returns the result in list form, of each item with the function applied. Using the following list...

```
(setq mylist ("apple" "orange" "banana" "zubrovka"))
```

You could use this code (imagine if there were hundreds of items in your list)...

```
(list (strcase (nth 0 mylist))  
      (strcase (nth 1 mylist))  
      (strcase (nth 2 mylist))  
      (strcase (nth 3 mylist))  
)
```

Or this code...

```
(mapcar 'strcase mylist)
```

To get this result...

```
("APPLE" "ORANGE" "BANANA" "ZUBROVKA")
```

When used with multiple lists, each list is processed in parallel one item at a time. Again, using these 2 lists...

```
(setq mylist1 '(1.0 2.0 3.0))  
(setq mylist2 '(0.1 0.2 0.3))
```

You could use this code...

```
(list (+ (car mylist1) (car mylist2))  
      (+ (cadr mylist1) (cadr mylist2))  
      (+ (caddr mylist1) (caddr mylist2))  
)
```

Or this code...

```
(mapcar '+ mylist1 mylist2)
```

To get this result...

```
(1.1 2.2 3.3)
```

But what if the 2 lists contained different numbers of items? (*mapcar*) only processes the lists while each list contains matching items in the other lists. In other words, it only processes the number of items in the shortest list and ignores (with no errors) any additional items the other lists may contain. Use this to your advantage. If we use two lists of unequal length like this...

```
(setq mylist1 '(1.0 2.0))  
(setq mylist2 '(0.1 0.2 0.3))
```

This code...

```
(mapcar '+ mylist1 mylist2)
```

Returns this result...

```
(1.1 2.2)
```

(lambda arguments expression)

You don't see this function by itself. It's usually wrapped inside another complex function like (*mapcar*) or (*apply*) and some of the Visual Lisp "VL-" based functions. To truly understand (*lambda*), it's best to think of it as being almost identical to (*defun*) which defines a function. When you use (*defun*) to create a function, the result is a function that you call many times throughout a program because the function is defined and stored in memory by name. Likewise, (*lambda*) also creates a function, but it's not named and not stored in memory, it's used on the spot, then and there without being able to be called again.

That's all fine and good but what does this do for you the programmer? What's it all mean? The essence of (*lambda*) is that it lets you define a function on the fly at the point in your code where it's needed, which often makes it more understandable, and without having to create the overhead in memory of actually defining a named function that you don't need anymore. Think of it this way...in Lisp programming, you create variables when you think you'll need the value again. If you don't need the value, you take the value and pass it to another function. Take the following two examples, both of which result in telling you how many seconds in a week which; we'll assume is the value we want to use. Would you use this code...

```
(setq seconds-per-minute 60)  
(setq minutes-per-hour 60)  
(setq hours-per-day 24)  
(setq days-per-week 7)  
(setq seconds-per-hour (* 60 60))  
(setq seconds-per-day (* seconds-per-hour hours-per-day))  
(setq seconds-per-week (* seconds-per-day days-per-week))
```

Or would you use this code...

```
(setq seconds-per-week (* 60 60 24 7))
```

Using (*lambda*) is essentially the same thing but on the function level instead of the variable level. You typically would not create a whole bunch of named functions all over the place (via *defun*) that you would never use again unless the function was large and complex or had a specific major purpose and objective. In these other cases, you can use the (*lambda*) construct instead. Let's

use our previous (*strcase*) example that we used with (*mapcar*) in the preceding section. (*mapcar*) worked real well with (*strcase*) when converting to lower case. But what about uppercase? (*strcase*) in that situation needs a second parameter, a T, to tell (*strcase*) to convert to lower case. For (*mapcar*) to work it needs something extra. Let's again use this data...

```
(setq mylist '("APPLE" "ORANGE" "BANANA" "ZUBROVKA"))
```

You could do something like this to pass the T along with the string to (*strcase*) by building a list of T's for each string value...

```
(mapcar 'strcase mylist '(T T T T))
```

This works well because as you learned earlier, each of the lists passes its items in parallel meaning a T goes along with each string to (*strcase*). However, you could also do this...

```
(mapcar '(lambda (x) (strcase x T)) mylist)
```

Doesn't seem much different does it. It's actually less clear using (*lambda*). But what if you don't know ahead of time how many strings are going to be passed? You'll have to create a loop that creates enough T's for that second list which further complicates the first example. In the second example that uses (*lambda*), it doesn't matter. We defined a function with (*lambda*) that adds the T. (*mapcar*) then just passes each item to the function, which is accessed with the function's argument X.

Also, consider that the first example worked well because we just blindly passed items from the lists to (*strcase*). But what if we needed to perform some type of operation on them? Mathematical operations, conditional testing (if/cond/eq/equal/...etc...) all would not have worked unless they were inside the (*lambda*) function, which could perform any number of operations in the data.

Let's take a look at another example. This time, we're going to give (*mapcar*) a list of Month names and we'd like back from (*mapcar*), a list of Month names abbreviated to the first 3 letters, with the initial letter being upper case and the last two lower case. Let's use this as our list...

```
(setq months '("JANUARY" "february" "March" "aPRIL" "MaY" "JUNE" "july" "AuGusT"  
"SEPtember" "octOBER" "NOVEMBER" "DeCemBer"))
```

With this data list, (*strcat*) can't do it. Neither can the (*substr*) function. Using them together along with (*strcat*), we can do it with the following code...

```
(mapcar '(lambda (x)  
  (strcat  
    (strcase (substr x 1 1))  
    (strcase (substr x 2 2) T)  
  )  
) months  
)
```

Which returns this as our list...

```
("Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec")
```

Now wasn't that much easier? You can even use multiple lists with *(lambda)*, just make sure there's enough argument variables declared so that there's one for each list that will be passing values from. Again, take this for an example...

```
(setq pt1 '(-3.24 2.82 0.0))  
(setq pt2 '(2.5 8.76))
```

With *(mapcar)* and *(lambda)*, we can easily find the midpoint between those 2 points using this code...

```
(mapcar '(lambda (c1 c2)  
          (/ (+ c1 c2) 2.0)  
        )  
        pt1  
        pt2  
)
```

This results in the point (-0.37 5.79) being returned. Notice that one of the points doesn't have a Z-coordinate yet the function doesn't error out. If you're working with 2d points, that's a good thing. If they both had a Z-coordinate, it would have worked with that as well. Now here's a variation of that same code you may see...

```
(mapcar (function  
          (lambda (c1 c2)  
            (/ (+ c1 c2) 2.0)  
        )  
        )  
        pt1  
        pt2  
)
```

In this example, *(mapcar)* isn't using a quoted function (*'*), instead it's using the *(function)* function. This was introduced because when running compiled AutoLISP code (FAS or VLX files), if the code fails in the right place in the right conditions, the quoted *(lambda)* function's source code could be exposed and displayed on the text screen. By wrapping it the *(function)* statement, it's protected from viewing during these types of failures. Using either form is perfectly fine, and one is not more correct than the other. Using the quoted function is a little easier to read and typically all that's needed if you aren't worried about keeping people from viewing your source code. Just as there's a quote function (*quote*) that can be used in place of the quote symbol (*'*), the *(function)* function does a similar thing at the function level. Now hopefully that wasn't too confusing with all the function functions and quote quotes.

Putting It All Together Back At The Office

When you can back to the office, try this for an exercise. This is the exact very same example I had years ago when I made the leap into using more advanced functions like *(mapcar)*, *(apply)*, and *(lambda)*. I'd seen those advanced functions used before, but I never took the time to use

them or figure them out let alone mixing several altogether nested. Then, I found the need to use the *(nentsel)* function, which returned nested data from a user selection. That is, I wanted information about a coordinate of an object within a block, not information about the block. The *(nentsel)* explanation in the AutoLISP reference explains the data it returns and how it's processed using the transformation matrix to get what you are looking for. The following code was what I came up with after 2 days where **pt1** was the point and **mat** was the transformation matrix as turned by *(nentsel)* ...

```
(setq pt1
  (list (+ (* (nth 0 (nth 0 mat)) (nth 0 pt1))
    (* (nth 0 (nth 1 mat)) (nth 1 pt1))
    (* (nth 0 (nth 2 mat)) (nth 2 pt1))
    (nth 0 (nth 3 mat))
  )
  (+ (* (nth 1 (nth 0 mat)) (nth 0 pt1))
    (* (nth 1 (nth 1 mat)) (nth 1 pt1))
    (* (nth 1 (nth 2 mat)) (nth 2 pt1))
    (nth 1 (nth 3 mat))
  )
  (+ (* (nth 2 (nth 0 mat)) (nth 0 pt1))
    (* (nth 2 (nth 1 mat)) (nth 1 pt1))
    (* (nth 2 (nth 2 mat)) (nth 2 pt1))
    (nth 2 (nth 3 mat))
  )
  )
)
```

It worked, but frustrated with the amount of effort and knowing that there must be a better way, I posted the question on the old ACAD forum on CompuServe (now part of AOL). I don't even recall whom the person was any more, but this is the code they posted back that did the same thing...

```
(setq pt (append pt (list 1.0)))
(mapcar '(lambda (x) (apply '+ x))
  (list (mapcar '* pt (mapcar 'car mat))
    (mapcar '* pt (mapcar 'cadr mat))
    (mapcar '* pt (mapcar 'caddr mat))
  )
)
```

As you can see, it's a lot smaller and more efficient. It was upon viewing that code, that I committed to breaking down this small snippet and figuring out exactly what it does and how it does it. When you're back at the office, try doing the same. Refer back to the *(nentsel)* documentation in the AutoLISP Reference so you know what the code is attempting to do. I guarantee, that when you go through this snippet and actually understand how it works and why, you'll never have trouble with *(apply)*, *(mapcar)*, or *(lambda)* ever again. Use the following blank lines to make notes as you disassemble this code and figure out what it does.

Recursive Programming – It's Kind Of Like Talking To Yourself

One of the oddest concepts that a lot of casual programmers have difficulty understanding, let alone implementing in code is known as Recursion. The simplest example of recursion is something most people have seen at one time or another. If you took a television camera and recorded the television itself, and played that to that same television you were recording, you'd see a recursive graphic. The television, would be displaying a picture of itself, displaying a picture of itself, displaying a picture of itself, etc.

Recursion in programming is essentially the same thing, it's when a program calls itself. Of course, unlike the television picture that goes on and on forever, with recursive programming, you need to have an "out" or your program would just go on forever in an endless loop until your software or computer ran out of memory.

It's a difficult concept to grasp beyond the basic definition of "a program that calls itself". Anyone can understand that, but wrapping your head around how you'd use such a technique in your programs is a little more mind numbing. So to get you thinking about it and understanding it, let's look at a few examples, which might get you thinking about how you can apply this from a programming perspective.

Suppose you built a machine that has one conveyor that fed into it, and one that fed out of it. The machine, removes cardboard boxes to reveal the contents of the box. Boxes move down the in feed conveyor, into the machine, the cardboard box is removed, and the contents come out on the exit conveyor. Now, what would happen if one of the boxes contained more cardboard boxes? As the machine processed the big box that had more, smaller cardboard boxes inside, it stopped accepting big boxes from the feed conveyor. It then took the smaller boxes, and fed those back into the machine again. The machine then stripped the cardboard off the small boxes, except one of those contained even more tiny boxes. The small boxes stopped feeding into the machine, and the tiny boxes went back in. When all the tiny boxes finished going into the machine, the small boxes started again, and when all those were done, the large boxes started again. This process that the machine we built was performing, can be thought of as recursion.

Now instead of a conveyor of boxes, think of data in a List. Maybe the List has strings, integers, real numbers. Maybe some of the items are Lists and some of those may or may not contain Lists. How would you process this data not knowing how deep it was nested? You'd build a recursive function that processed the List, and when it encountered a nested List inside of the List, it would halt processing the List (temporarily), while it called itself and handled the nested List and when finished with the nested List, it would continue with the main list.

Another good example is a program that processes folders on your hard drive and finds files. You can point it to a folder, but does that folder contain more folders? And do those folders contain more folders? And are files scattered throughout at all levels within those "unknown" levels of sub-folders? This would be a good case for using recursion. This is the essence of recursive programming. Anytime you have an unknown set of potentially nested conditions or processes you need to perform, or data you need to examine, recursive programming can come to the rescue.

Let's start looking at some code. To begin with, we'll write code that returns the factorial of the number 4. If you're unfamiliar with factorials, it's the mathematical process where you take a number, multiply it by a number one less, multiplied by a number one less until you get to 1. So, the factorial for the number 4 is the following formula $4 * 3 * 2 * 1 = 24$.

```
(defun factorial (n)
  (cond
    ((= n 0) 1)
    (t (* n (factorial (1- n))))
  )
)
```

Recursion is a good option here because we don't always know how many times we'll need to multiply unless we know the number beforehand. In this case we do and it's 4 so we call the function with the syntax (factorial 4). Now, while our formula is $4 * 3 * 2 * 1$, this program takes a slightly different approach, it works from the opposite direction, it's actually calculating $1 * 2 * 3 * 4$. If you think about it, that's what all of your AutoLISP programs do, they work into the nested most functions, then slowly work back.

In this case, the program doesn't try to perform $(4 * 3)$, it tries to perform $(4 * (\text{factorial of } 3))$. But to calculate the factorial of 3, it needs to multiply it by the factorial of 2, which it also doesn't know and so on. What might help you understand a little better, is to list the formula so it reflects more closely what's going on is this...

```
Factorial of 4 = (* 4 (Factorial of 3))
Factorial of 3 = (* 3 (Factorial of 2))
Factorial of 2 = (* 2 (Factorial of 1))
Factorial of 1 = (1 * 1)
```

Or putting it all together in an AutoLISP like syntax it looks like this...

```
(factorial 4) = (* 4 (* 3 (* 2 (* 1 1))))
```

Here's some code that reflects this without using recursion (without the program calling itself) ...

```
(defun factorial4 (n)
  (* n (factorial3 (1- n))))
)
(defun factorial3 (n)
  (* n (factorial2 (1- n))))
)
(defun factorial2 (n)
  (* n (factorial1 (1- n))))
)
(defun factorial1 (n)
  n
)
```

Calling the above code (`factorial4 4`) will do the same thing as the recursive example. Except in this case, instead of calling itself, it calls a completely different function that's almost identical. As you can see, a lot of the code is the same. And the "unknown" nature of not knowing how many times (how deeply it's nested) makes this a good application for recursion.

The fact that we're dealing with numbers in this example, means that it would really not be that hard to create a loop with a few counter variables to perform this same calculation. But not all cases would be that easy. Let's look at another example that deals with nested lists...

```
(setq mylist '(1 2 3 (4 5 (6 7 (8 9) 10) 11) 12 13 (14 15 (16))))
(defun item-in-list (L I)
  (vl-some '(lambda (x)
              (cond
                ((= x I) T)
                ((= (type x) 'LIST) (item-in-list x I))
                (T nil)
              )
            )
    L
  )
)
```

The code in this example looks for a user specified item within a list. The AutoLISP function (`member`) typically handles this type of need. But if the list contains other nested lists, (`member`) just doesn't do it anymore. In this case, we're using a function called (`vl-some`). This function takes a quoted function, just like (`apply`) or (`mapcar`). The purpose of (`vl-some`) is to return T if the function evaluates to T for at least 1 (one) item in the list. If the function evaluates to NIL for all items, it returns NIL. It's the perfect function for our needs. Next, we build our function. Just like (`mapcar`), (`vl-some`) will be handed each item in the list, one at a time. Your function then needs to check if that item matches what we're looking for. That's not hard. But what if the list item, is another list? Our function must also check for that, and if the list item is another list, it calls itself again to process the nested list. Now, (`vl-some`) itself isn't looking for our item, our (`lambda`) function is doing that. What (`vl-some`) does is looks for T or NIL and it returns T or NIL. (`lambda`) is pasting the T and/or NILs to (`vl-some`).

When processing a nested list that contains our item, (`vl-some`) returns T but it returns it to itself because it was called from a function it was using. And because a T was returned to it, it then returns a T. So with our example, calling (`item-in-list mylist 13`) would return T while calling (`item-in-list mylist "Cat"`) would return NIL because it's not contained in any of the data in the list or nested lists.

Now that you've seen a couple examples, it's really not that hard. In fact, the hardest part of writing a program that uses recursion, is to make sure the program has an "out" or at some point stops processing and doesn't just blindly call itself forever. But it takes a while; you won't be an expert just by reading this. You'll become an expert only after using this technique many times. To help you with this, here's a little sample code. This code was posted to the AutoCAD's Customization discussion group years ago. It was created by Owen Wengerd (<http://www.manusoft.com>) and is a very good example of short, well-written recursive program.

```
(defun Str+ (source / prefix rchar)
  (cond
    ((= source "") "A")
    ((= (setq prefix (substr source 1 (1- (strlen source)))
      rchar (substr source (strlen source))
    )
      "Z"
    )
      (strcat (Str+ prefix) "A")
    )
    ((strcat prefix (chr (1+ (ascii rchar))))))
  )
)
```

What this code does, is it takes a string character and increments it. Its intent is to generate the next revision letter based in the current revision. So let's say your drawing is at revision "B", calling the function (str+ "B") would return "C". Calling the program with (str+ "C") would return "D" and so on. That's not that hard in itself, and you wouldn't need recursion to perform it. But what happens when you get to revision "Z" and need to go to "AA" or you're at revision "AZ" and need to go to "BA". This program handles that and that's where the recursion comes in. So your homework when you get back to your office is to slice and dice this program up and see how it works. Play with it a little and see if you can truly understand what it's doing. And if you're really ambitious, try modifying the program so that it works as described, but does Not use the letters "O" (oh) and "I" (eye) which resemble 0 (zero) and 1 (one).

Visual LISP Primer

Why Visual LISP? – 2905 Reasons, That's Why!

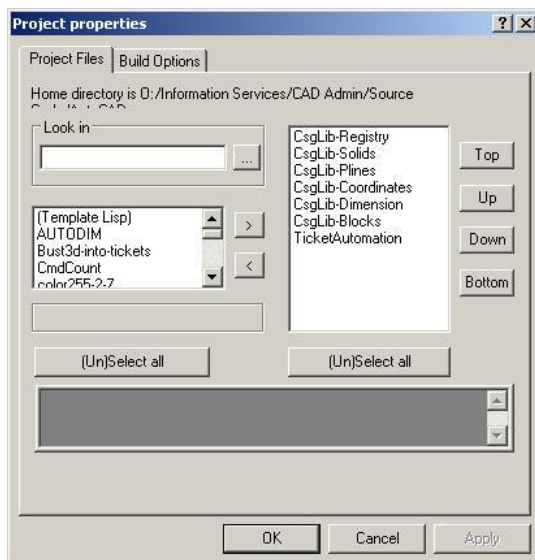
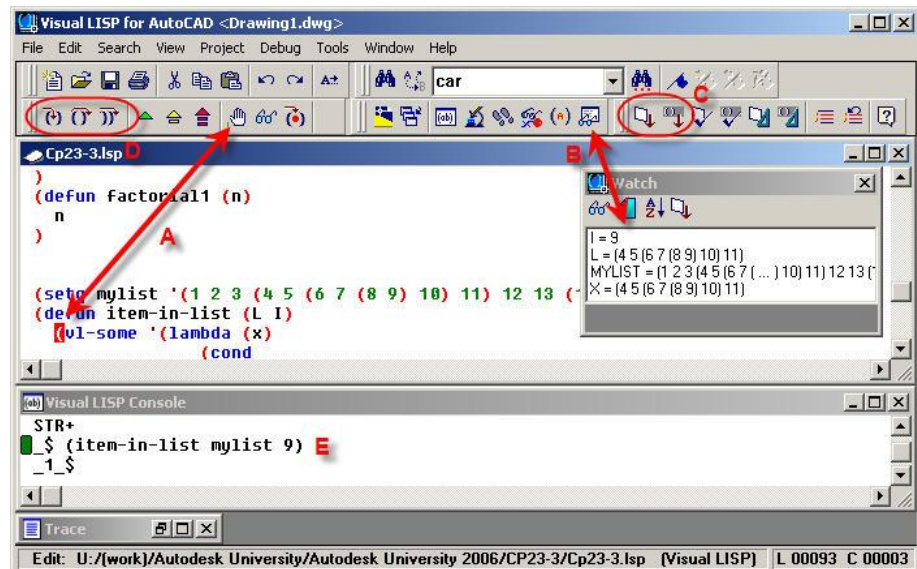
If you've ever wondered what's in Visual LISP that might make it attractive for you to use, here's a little teaser to peek your interest.

- 80 Visual LISP function (those prefixed with VL-)
- 2042 ActiveX function specific to AutoCAD's object model. (those beginning with "vla-" prefix)
- 47 Reactor functions (those beginning with "vlr-" prefix)
- 121 Reactor types or event triggers (those beginning with "vlr-" prefix)
- 69 Generic ActiveX functions (those beginning with "vlax-" prefix)
- 1309 ActiveX constants (those beginning with "ac", "acrx-", and "vlax-")

Of course there are more than just additional functions that make it attractive. For starters, you can step through your code line-by-line, watching where control flows and the values of variables as they are changed. Look at **A** in the following image. You can add breakpoints to your code then use one of the buttons marked **C** to load the code with breakpoint information into memory. At the bottom of the screen **E**, you can then call your code and it will pause at the breakpoint. The

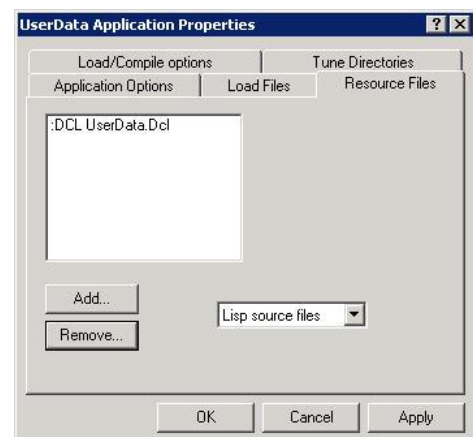
Watch Window **B** then displays your variable's values (you need to add which ones you want to watch) as you manually step through the code using the buttons marked **D**.

When you use the Visual LISP editor it makes troubleshooting your code a lot easier with this ability. You can easily see where, when as well as why some variables are not containing the information you think they are which in turn allows you to more quickly fix the issue. Of course, there are still other benefits to using the Visual LISP editor. As you likely already know, it allows you to build compiled AutoLISP files such as FAS (Fast load AutoLISP Source) and VLX (Visual Lisp Applications). Contrary to popular belief, there are a lot of reasons to compile your projects other than to protect the source code.



Solids, Plines, Coordinates, Dimensions and Blocks. This code isn't copied and pasted to the new AutoLISP file, it's just referenced and then compiled into the same FAS file (or separate if you like). As more functions are added to the library code or the existing ones updated, a quick recompile of the project, and any other projects that use that library code are then updated. It makes code sharing and reuse a lot easier.

One of the benefits of creating FAS files is that you can build them from a project. And a project, as shown in the image to the left, can have a number of AutoLISP files defined in it. What does this do for you? You can more easily build libraries of reusable utility functions that you can access from your code. In the example to the left, the TicketAutomation project is referencing utilities that work with the Registry,



Still another advantage of the Visual LISP editor is the ability to compile Applications (VLX). Similar to Projects which build FAS files, Visual LISP

applications offer a lot more options, including the ability to include within them, other projects, Lisp source code, compiled Lisp code (FAS), text files and even DCL dialog files. All this within one file makes distributing more complex programs and applications a lot simpler. There's even an option to compile the Visual LISP Application in such a way that it runs in its own namespace which means all the code and variables run in its own protected memory space. If you've ever had an AutoLISP program fail because another program used the same variable name and it was set to something different than your program expected, compiling an application to run in a separate memory name space would solve this problem.

There's simply a lot in the Visual LISP editor, which makes it very worth your while to learn how to use it. Unfortunately, there's just too much to cover completely in this course, which is why we've just covered some of the major highlights that can really affect your coding productivity.

Reactors – Code That Knows Not Just What To Do, But When To Do It

Another nice thing about Visual LISP is the ability to create Reactors. Reactors are event triggers. That is, when certain “events” happen within your drawing or AutoCAD, the “trigger” calls some code or action. You, the programmer, define all the Events and the Actions that the Triggers call. Reactors aren't that difficult to master, but there are a few things to know so here's a rundown.

First, there are two types of Reactors, Transient and Persistent. Transient reactors are not saved with the drawing and are the default method when Reactors are created. If a Reactor is transient, and you want it to apply to a drawing all the time, you need to add it each time the drawing is loaded. If you want a Persistent reactor, you need to add a reactor (it'll be transient) then convert it to Persistent. Persistent reactors are saved in the drawing and do not need to be added again unless they are intentionally removed with other AutoLISP code.

Second, regardless if a Reactor is transient or persistent, know that the reactor itself is just a trigger that calls an AutoLISP function. If the functions the Reactors are supposed to call are not loaded, the reactors will give the user an error. For this reason, transient reactors are typically all that's needed in most cases because if you make considerations for loading the code the reactors call, you might just as well load the reactor anyway as well.

Third, when you review the VLR- functions in the AutoLISP reference, you will see some function names that refer to “Reactors” and a few that refer to “Reactions”. There are 19 Reactors and each Reactor has one or more Events. These Events are what the Reaction functions are talking about. For example, there is a *:vlr-DWG-Reactor* that deals with things that happen to the drawing database. That particular Reactor has 8 Events or Reactions that it can use which are *:vlr-beginClose*, *:vlr-databaseConstructed*, *:vlr-databaseToBeDestroyed*, *:vlr-beginDwgOpen*, *:vlr-endDwgOpen*, *:vlr-dwgfileOpened*, *:vlr-beginSave*, and *:vlr-saveComplete*.

Fourth, the callback function is the function the reactor calls. Each callback function takes two arguments, one (the first) holds the event name that ended up calling the code, the other contains any other data (if any) that that particular event may pass back. The event *:vlr-commandEnded* for the *:vlr-command-reactor* for example, will pass a string indicating the command name that ended.

Now that those issues have been addressed, let's look at some example code. The following code needs to be run manually. The function (*save-pre*) and (*save-post*) are callback functions that will

be called when you save a drawing. The command (*c:Load-My-Reactor*) will create different event reactors for the *:vlr-DWG-Reactor* reactor. Once you load the AutoLISP code into memory, run at the command line *Load-My-Reactor*. Nothing will appear to happen but something actually did, the Reactor had 2 different event triggers added to your drawing. When you save your drawing, you'll be alerted pre-save and post-save with an alert box.

;;; Load Up The Reactor

```
(defun c:Load-my-reactor ( / myreactor)
  (vl-load-com)
  (setq myreactor (vlr-dwg-reactor
    "This is my SAVE Reactor"
    '(:vlr-beginSave . save-pre)
    (:vlr-savecomplete . save-post)
  )
)
(vlr-add myreactor)
)
```

;;; Pre-Save Reactor Callback

```
(defun save-pre (reactor data)
  (alert (strcat "Drawing is about to be saved by "
    (getvar "loginname")
  )
)
)
```

;;; Post Save Reactor Callback

```
(defun save-post (reactor data)
  (alert
    (strcat "Drawing has been saved by " (getvar "loginname"))
  )
)
```

Now, because these Reactors are transient, if you close and reopen your drawing, future drawing saves will not show the alerts unless you recreate the reactors again by running the code again. If you want a reactor to be present all the time in your drawing file, you will want to add code to your ACAD/ACADDOC.LSP file for this purpose.

Now, something that's less known, is that you can add the same reactor more than once. If you load the code above and run the command *Load-My-Reactor* several times, when you save your drawing you'll get several messages. Typically, if you are adding reactors upon drawing open, this isn't a big issue. But if you have code that's manually initiated by a user, like in this example, it's possible to add the reactor more than once, which may cause you problems. For this reason, it's wise to verify if the reactor is loaded already first, before adding it to your drawing. The following 2 functions will do this for you.

;;; VLisp Reactor - Is Reactor Loaded ?

```

;;; Call: (au:vlr-loaded-p fname {rtype} {revnt})
;;; Return: T/nil
;;;
;;; _____
;;; fname = Function name / 'function
;;; rtype = Reactor type / :vlr-xxxx-Reactor
;;; revnt = Reactor event / :vlr-xxx
;;;
;;; _____
;;; Dependent Functions...
;;; au:vlr-fun-eq-cback
;;;
;;; _____
(defun au:vlr-loaded-p (fname rtype revnt / au$reactors)
  (if fname
    (progn
      ;; Get List of Reactor Types w/Reactors Loaded - If Any
      (if (vl-member-if
          '(lambda (i) (eq i rtype))
          (vlr-types)
        )
        (setq au$reactors (vlr-reactors rtype))
        (setq au$reactors (vlr-reactors)))
      )
      ;; Check If Reactor Callback Loaded
      (vl-some
        '(lambda (i)
          ;; Check If Reactor Callback Loaded For Any Reactor
          (vl-some '(lambda (ii)
            ;; Check If Reactor Callback Loaded For Each Reactor
            (au:vlr-fun-eq-cback fname ii revnt)
          )
            i
          )
        )
        )
      (mapcar 'cdr au$reactors)
    )
    nil
  )
)

;;; VLisp Reactor - Is Function Equal To Callback
;;;
;;; _____
;;; (au:vlr-fun-eq-cback 'fname robj {revnt})
;;;
;;; _____
;;; fname = Function name      / 'function
;;; robj = Reactor object      / reactor
;;; revnt = Reactor callback event / :vlr-?

```



```

;;;
;;; Dependent Functions...
;;; N/A
;;;
(defun au:vlr-fun-eq-cback (fname robj revnt / au$revnt-list)
  ;; Get Reactions
  (setq au$revnt-list (vlr-reactions robj))
  ;; If Event Specified, Restrict To Event - Else Look At All
  (if revnt
    ;; Restrict To Event
    ;; If Function Name = Reactor Callback
    (if (eq fname (cdr (assoc revnt au$revnt-list)))
      T
      nil)
    )
  ;; Else Look At all
  (vl-some '(lambda (i)
    (eq fname (cdr i))
  )
    au$revnt-list
  )
)
)
)

```

The explanation of all that these two functions do and how they do it is beyond what we have time for here in this class. However, note that the function (*au:vlr-loaded-p*) is the only function you need to call. The second function is used only by the first. The comments at the heading of the function explain how to call it, but essentially, you call the function and pass the quoted callback function, the reactor, and the event/reaction as parameters and the code will then return T if that function is already associated with that event or that reactor. If it's not, then it will return NIL. With that in mind, we can modify the *Load-My-Reactor* code to the following to keep it from loading the reactors more than once.

```

;;; Load Up The Reactor (load only once)
(defun c:Load-my-reactor (/ myreactor)
  (if
    (and (not (au:vlr-loaded-p
      'save-pre
      :vlr-dwg-reactor
      :vlr-beginsave
    )
    )
    (not (au:vlr-loaded-p
      'save-post
      :vlr-dwg-reactor
      :vlr-savecomplete
    )
  )
)

```

```

    )
  )
  (progn
    (vl-load-com)
    (setq myreactor (vlr-dwg-reactor
      "This is my SAVE Reactor"
      '(:vlr-beginSave . save-pre)
      (:vlr-savecomplete . save-post)
    )
  )
  (vlr-add myreactor)
)
)
)

```

Some other things that are important to know about reactors are that you shouldn't prompt for user input or call the *(command)* function in a reactor's callback function. While it can be done successfully in some cases, it can also cause problems because some reactors fire while commands are still active. Don't count on being able to do this and be prepared for problems. Also note that just like with recursive programming, it's possible to have a reactor fire itself. That is, if your save reactor saved the drawing, or if an object reactor modified an object, it could fire itself resulting in an endless loop. You'll need to make programming considerations in your code to check for those conditions to prevent them from occurring. The most common method is to have your callback function check for the value in a variable, and if it's not set, set it and proceed with the callback function. If the event triggers itself, it'll see that the variable is already set and not finish. Just don't forget to clear the flag when your code is finished processing whatever it's doing.

There's a lot to reactors. Review the AutoLISP reference for the various types of reactors available as well as their events and the various functions that help you manage reactors. If you're ambitious, dissect the 2 utility functions I've supplied and see how they work. If you can understand the logic and workings of those 2 utility functions, you'll have already mastered programming reactors in Visual LISP.

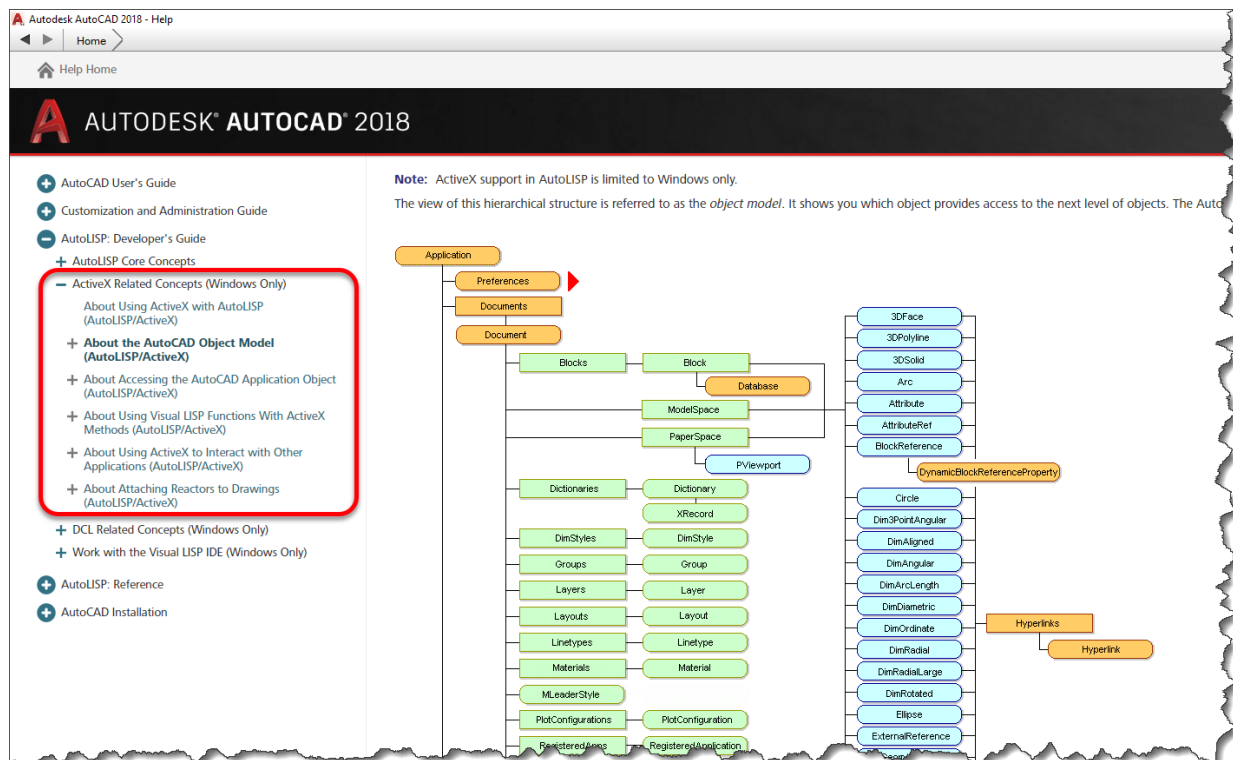
ActiveX – The Power Of Visual Basic In Visual LISP

There are a number of additional functions in Visual LISP that provide ActiveX functionality inside of AutoCAD and other applications. These functions aren't initially available inside Visual LISP unless support for them has been explicitly loaded. You can load and access these ActiveX functions easily with the Visual LISP function *(vl-load-com)*.

These ActiveX functions provide access to the methods, and properties of the AutoCAD object model as well as supporting functions and those that provide access to the object models of other applications that support ActiveX. If you browse through the AutoLISP Reference (in AutoCAD's Developer Help), you'll see the reactor (*vlr-*) and (*vlax-*) based ActiveX functions listed. The *vlax-* based functions are more generic in nature and therefore covered fairly well in the AutoLISP

Reference. The *vla-* functions on the other hand, are a different story. You won't find a list of them or documentation of them anywhere in AutoCAD's help system. In fact, there's likely not a complete list or explanation of the *vla-* Visual LISP functions anywhere on the planet other than the smaller lists of certain functions that people have put in CAD books or on their CAD based web sites. Nowhere have I found a complete list of them with explanations let alone sample code. Just looking at the sheer number of the functions involved, this likely explains why that is. The question then comes to mind, if these ActiveX Visual LISP functions aren't listed anywhere, then how does someone find out about them, let alone figure out how to use them? You'll find out how in this section of this course.

To start with, you'll need to be familiar with what's referred to as AutoCAD's ActiveX "Object Model". It's a tree like structure that holds everything related to AutoCAD. In essence, it's a family tree for all things AutoCAD. Other software programs that support ActiveX automation have object models too. This "Object Model" is how an ActiveX based application defines the database structure of AutoCAD and defines its properties and methods. Oh, yea, I almost forgot to tell you, "Methods" are the term used for "functions" when dealing with ActiveX automation.

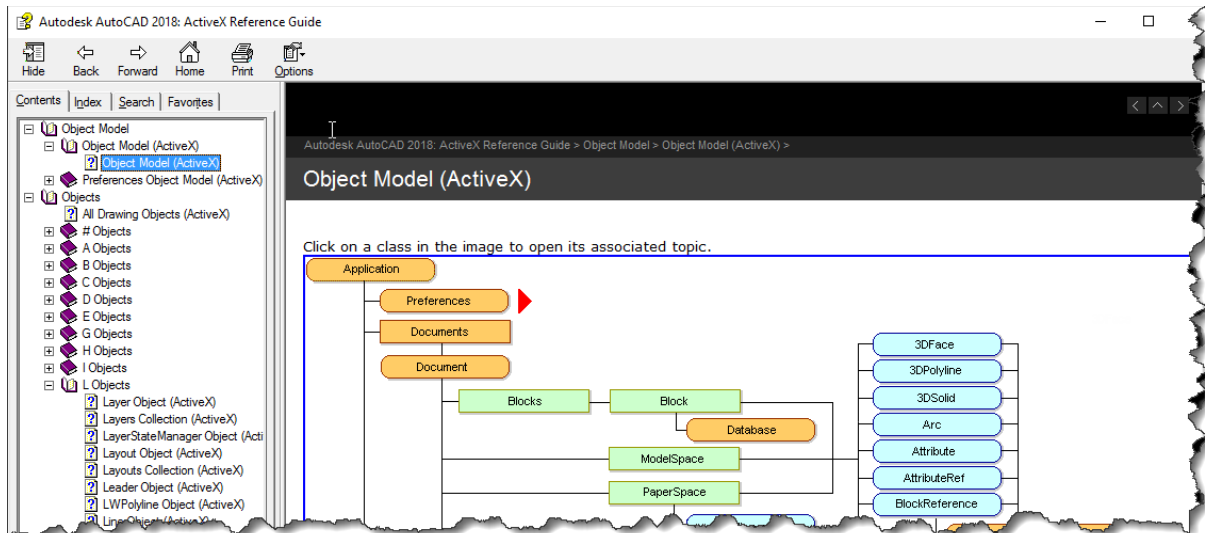


To give you an idea of what an object model looks like, you'll have to go to AutoCAD's Help System. Navigate to the ActiveX concepts in the AutoLISP Developer Guide as shown above to view AutoCAD's Object Model. The links here tell you a lot about how to interact with the ActiveX Object Model using AutoLISP.

Once you start trying to write code, you'll also want to access a different set of more robust documentation. You'll be accessing this information almost constantly if you're going to be doing ActiveX automation from Visual LISP until you become more familiar with its use. To access this

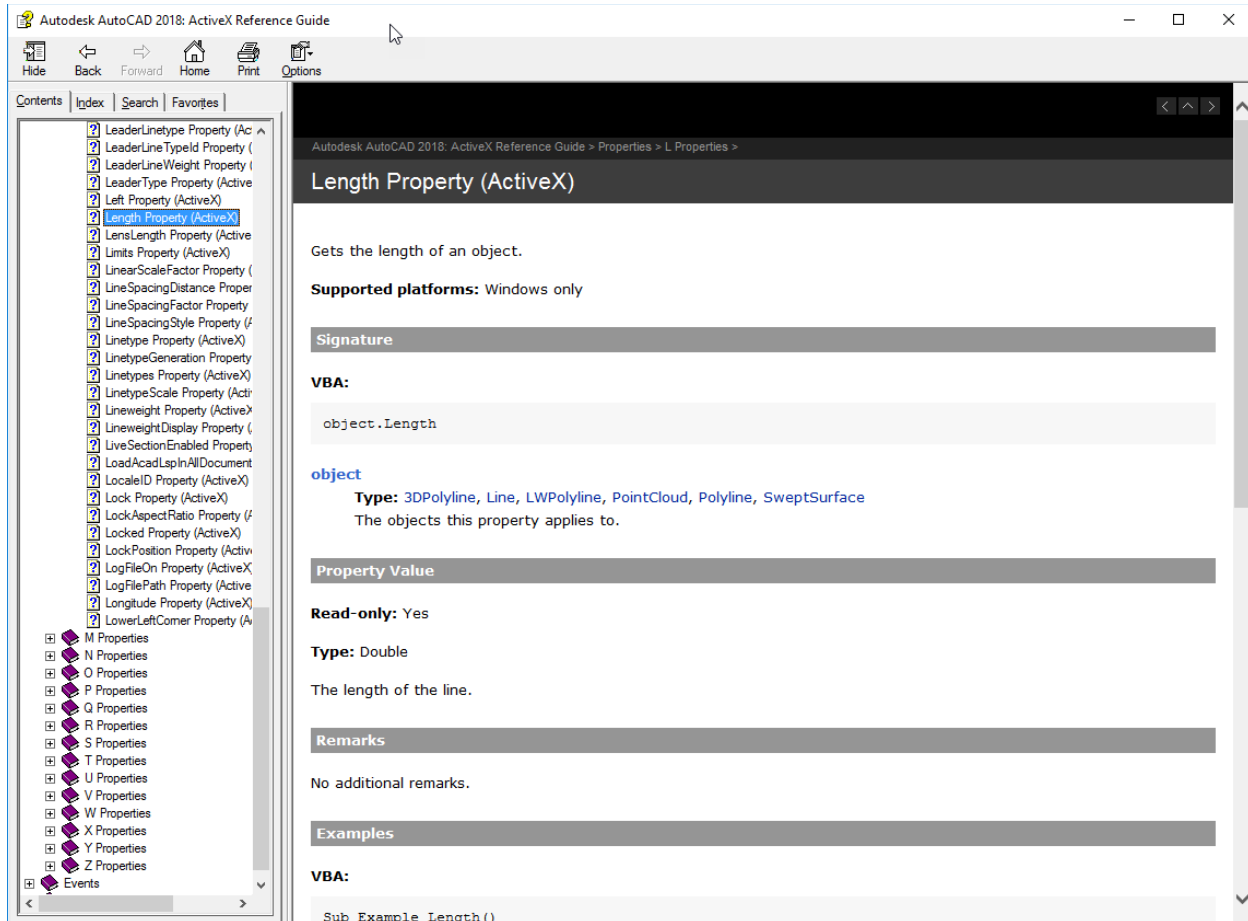
documentation, go and click on the file **C:\Program Files\Common Files\Autodesk Shared\en-US\acaauto.chm**.

You can get a similar view of the Object Model as shown above you also have access to additional documentation of each object and its methods, properties and events that the on-line help won't give you.



When you get to the Object Model in this version of Help, clicking on any of the items in the Object Model will display information for that item. You'll also notice by looking at the object model that there are rectangular shapes with square corners and rectangular shapes with rounded corners. Looking at the legend, you can see that those with square corners are a *Collection* and those with rounded corners are an *Object*.

A *Collection* is really a type of object as well but it's special in that it really doesn't do much other than contain and group other objects. Every *Layer* in AutoCAD is stored and accessed in the *Layers* collection. Every *Block* is stored and accessed in the *Blocks* collection and every drawing entity (line, arc, circle, text, dimension, etc.) is stored and accessed in either the *ModelSpace*, *PaperSpace* collection or the *Block* object.

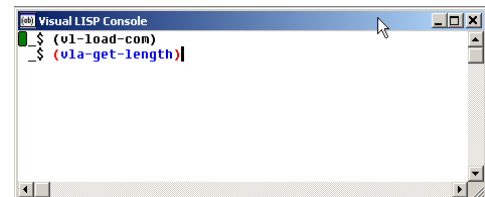


So to start, let's look at a *Line* in the Object Model (*Application* -> *Documents* -> *Document* -> *ModelSpace* -> *Line*). If you look at the help for the *Line* object, you'll see several *Methods*, *Properties* and *Events* listed. If you look at the *Properties* specifically, you'll see one of those is for *Length*. From AutoCAD's Object Model, you can directly extract a *Line*'s *Length* instead of calculating it with the (*distance*) function in AutoLISP. You'll find a lot of these useful properties for the various entities in AutoCAD that you previously needed to calculate yourself in AutoLISP. But now with the ActiveX functionality in Visual LISP, you can get access to a lot of this information quicker than ever before.

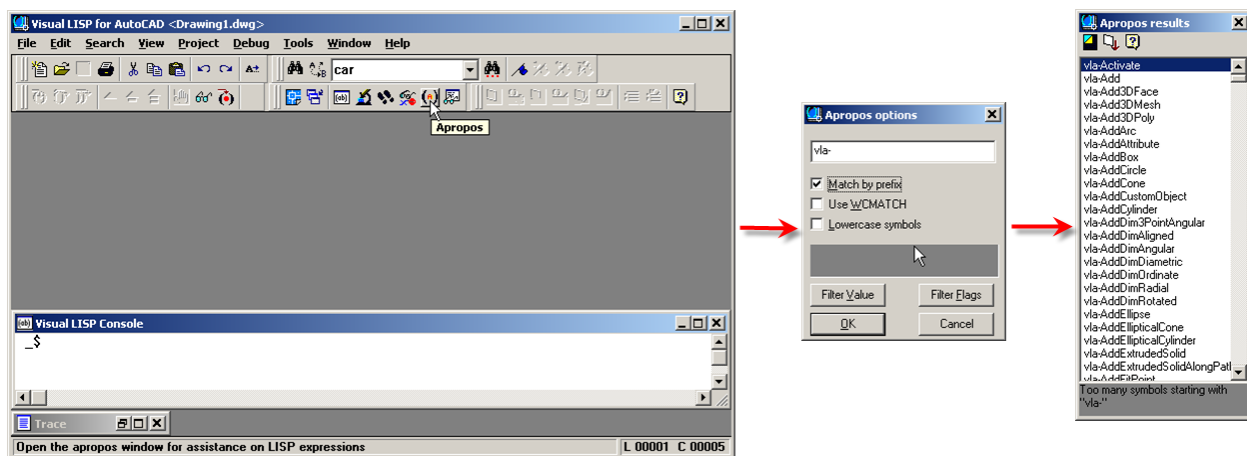
But the question still remains, how do you take this information in the Visual Basic documentation and turn it into an ActiveX function in Visual LISP? The answer is that you already have all the information you need; you just need to know how to translate it. And the key is this... Visual LISP ActiveX functions specific to the object model take on a predictable format. For starters, they all start with "vla-". Next, are you extracting a property or setting a property? If you are extracting a property, the next part of the function is "get-" and if you are setting a property, the next part of the function is "put-". And finally, the last part of the function name is the property name in VBA. In this example, that's "length". So, to extract the *length* of a line, the name of our Visual LISP function would be (vla-get-length).

If you look at the documentation, you'll see that not only is there a VBA example, but there's an AutoLISP example as well. Also notice in the documentation, *Length* is a Read-Only property. That means you can't "set" the property and therefore there isn't a (*vla-put-length*) function in Visual LISP. If the property was not Read-Only, there would be a corresponding *vla-put-<VBA Property>* function. This format is the same for ALL properties you'll find in the ActiveX and VBA Reference.

With this format information, you now know how to figure out the ActiveX Visual LISP function for every property in the AutoCAD Object Model. Very easy don't you think? To make sure you have the proper function name, you can easily test this by typing the function in the Visual LISP editor, making note of its color. Don't forget to call (*vl-load-com*) function first. Typically, you would call (*vl-load-com*) once in every AutoLISP program you write that uses ActiveX but in reality, once it's called, it stays loaded for the entire AutoCAD session.



Now that you know how to figure out the ActiveX functions in Visual LISP for getting or setting an ActiveX property, the next thing to figure out is what the names of the Visual LISP functions are for ActiveX methods. This information you also had previously. If you look back at the help information for the *Line* object, you'll see that it says it can be created with the *AddLine* method. This is the key to figuring out the function name. Again, the beginning of any AutoCAD Object Model specific function in Visual LISP starts with "vla-". Only this time, you just add the method name. This would make the ActiveX Visual LISP function to create a line (*vla-addline*). This format holds true for ALL ActiveX methods in Visual LISP, they are named in exactly the same way. Again, this can be verified by typing it in the Visual LISP editor, making note of the color of the function name.



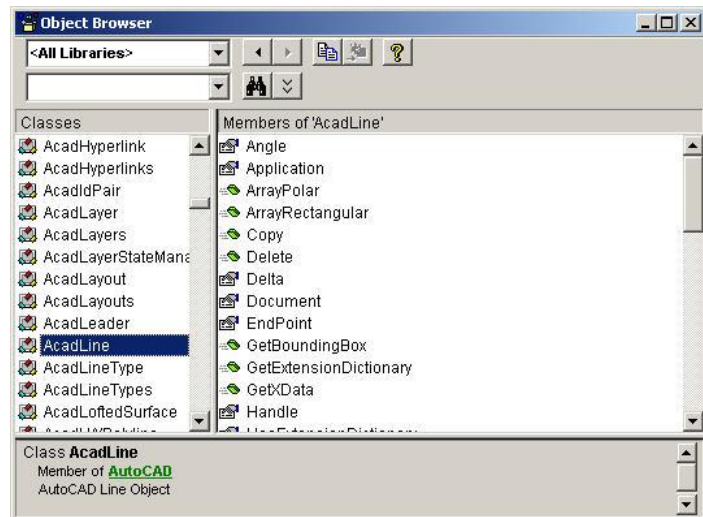
Now you're armed with all the information you need to figure out what all the 2042 different Visual LISP ActiveX functions are. The only thing left if to figure out how to use them but that will come shortly. Next, we'll explore a couple different ways other than the ActiveX and VBA Reference to

find these types of function names. For starters, you can list all of these functions right from within the Visual LISP editor. To do this, all you need is the *Apropos* function in the editor.

You'll notice that there's just too many entries to list all at once so to see all of them you'll have to search for prefix strings like "*vla-a*" or "*vla-r*". This can come in handy, after you've used ActiveX Visual LISP functions for a while and you already know how to call them but just need a quick refresher on their spelling or need to quickly browse for a function that you know exists but don't quite recall how it was spelled.

There's also another easy way to browse this type of information in its Visual Basic format. To do this you have to first install the VBA model for AutoCAD as it no longer ships out of the box. Simply go to <http://www.autodesk.com/vba-download> and you'll find links to download and install the VBA module.

Once installed, start the VBA editor by typing "VBAIDE" in AutoCAD (not the Visual LISP IDE) Once in the VBA editor, open the "Object Browser" by selecting *View -> Object Browser* from the pull down menu.



The Visual Basic editor opens the Object Browser with it set to display the AutoCAD Object Model library. Using this object browser won't help any more than using the ActiveX Reference, it's just another way to access the same information.

Now that you know of several ways to find Visual Basic methods and properties and convert that data into a Visual LISP function, the next step is to figure out how to call them. This is the most difficult part by far of working with ActiveX objects in Visual LISP. The first task will be to get the length of a line in AutoCAD with the (*vla-get-length*) function that we determined we needed earlier to extract this type of information.

If you look at the Help section for this, you'll see the signature for using this property is *object.length*. This is how this property is used in Visual Basic. *Object* is the AutoCAD object that you're using the property on and *Length* is the property you're using on that object. In Visual Basic, there are often parameters associated with methods. Properties on the other hand are generally just set with an "=" in Visual Basic. In the case of the Read-Only *Length* property, there are no arguments that go along with it in Visual Basic. The only information it needs, is the object that is specified prior to the property and separated with a period. In Visual LISP, this object needs to be specified as an argument (parameter). It's always the first argument in the function. If the property or method as listed in the ActiveX and VBA Reference has additional arguments or values to set, they are listed after the object argument in the same order that they appear in the ActiveX and VBA Reference documentation.

To extract the length of a line, you would call the (*vla-get-length*) function passing to it a *Line* object. Because you are using ActiveX methods and properties, your arguments need to be

ActiveX arguments as well. This means that you can't just pass an entity name to the function like you are used to doing in AutoLISP but instead it must be converted to an ActiveX object. This is done with one of the more generic ActiveX Visual LISP functions (*vla-ename->vla-object*). Here's how it would look...

```
Visual LISP Console
_ $ (vl-load-com)
_ $ (setq myline (car (entsel)))
<Entity name: 40075d58>
_ $ (setq mylineobj (vla-ename->vla-object myline))
#<VLA-OBJECT IAcadLine 00F43Fb4>
_ $ (if (and (vla-property-available-p mylineobj 'Layer)
            (vla-write-enabled-p mylineobj))
        )
        (vla-put-layer mylineobj "Wall")
)
nil
_ $ |
```

```
Visual LISP Console
_ $ (vl-load-com)
_ $ (setq myline (car (entsel)))
<Entity name: 40075d58>
_ $ (setq mylineobj (vla-ename->vla-object myline))
#<VLA-OBJECT IAcadLine 00F43Fb4>
_ $ (vla-get-length mylineobj)
1.25595
_ $
```

Notice that we first called (*vl-load-com*). If this was done previously in this session of AutoCAD, there's no need to do it again. If you do call it again, it does nothing so calling it when it's not needed is not a problem. Next, we selected a line and determined its entity name. It was then

converted to an ActiveX *Line Object* with the help of the (*vla-ename->vla-object*) function. Once that was done, we could then use the (*vla-get-length*) function to extract the length of the line with the function being called in the in the following format... (***vla-get-length lineobject***).

As any good programmer knows, error checking is always an important factor. With that in mind, what if the operator selected something that didn't have a *length* property like a piece of Text? The answer is by using the (*vla-property-available-p*) function that is documented in the Visual LISP Reference. This will tell us if the property we want to use is valid for the object selected. This function requires the Line object and the property quoted as an argument. Here's how this would look with this type of error checking...

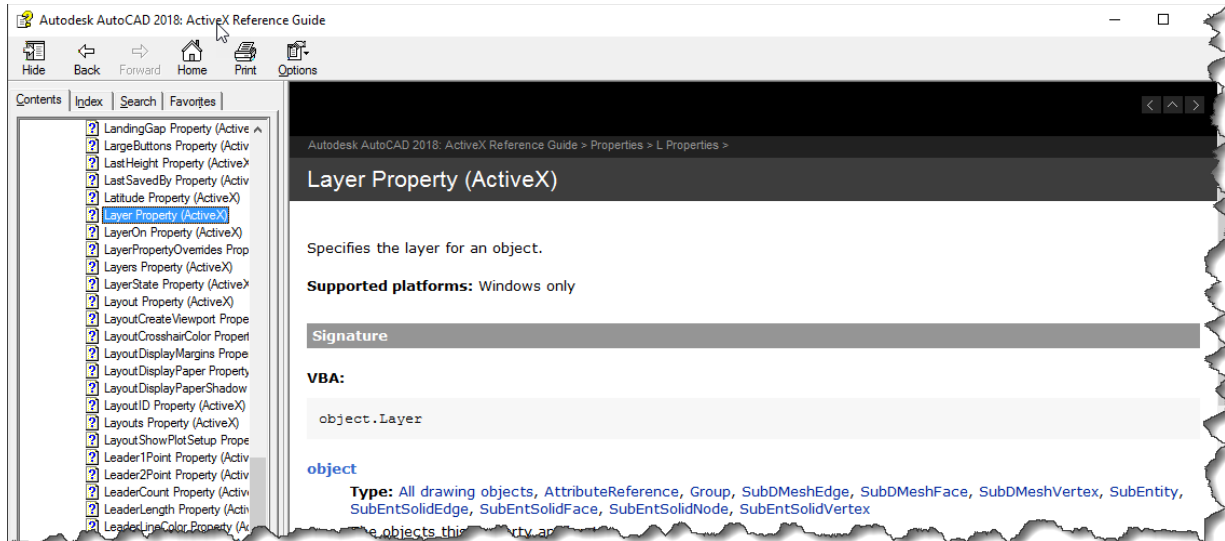
```
Visual LISP Console
_ $ (vl-load-com)
_ $ (setq myline (car (entsel)))
<Entity name: 40075d58>
_ $ (setq mylineobj (vla-ename->vla-object myline))
#<VLA-OBJECT IAcadLine 00F43Fb4>
_ $ (if (vla-property-available-p mylineobj 'Length)
        (> (setq mylength (vla-get-length mylineobj))
            (> (alert "Error: No line selected.")
            (> )
            1.25595
            _ $
```

You'll see that by first testing what we think is a Line object; we can make certain that a *Length* property is indeed available for the selected object. You should also start noticing that the *vla*-functions go hand in hand with the *vla*- functions.

Now that we can extract a property, let's try setting one. This property will be the Layer property. Using the function naming convention, we outlined earlier, the function would start with "*vla*". The next part of the function would be "*put*-" because we are setting a property or "Putting" a value in that property. Finally, the property itself is "*Layer*". This would make the Visual LISP ActiveX function (*vla-put-layer*).

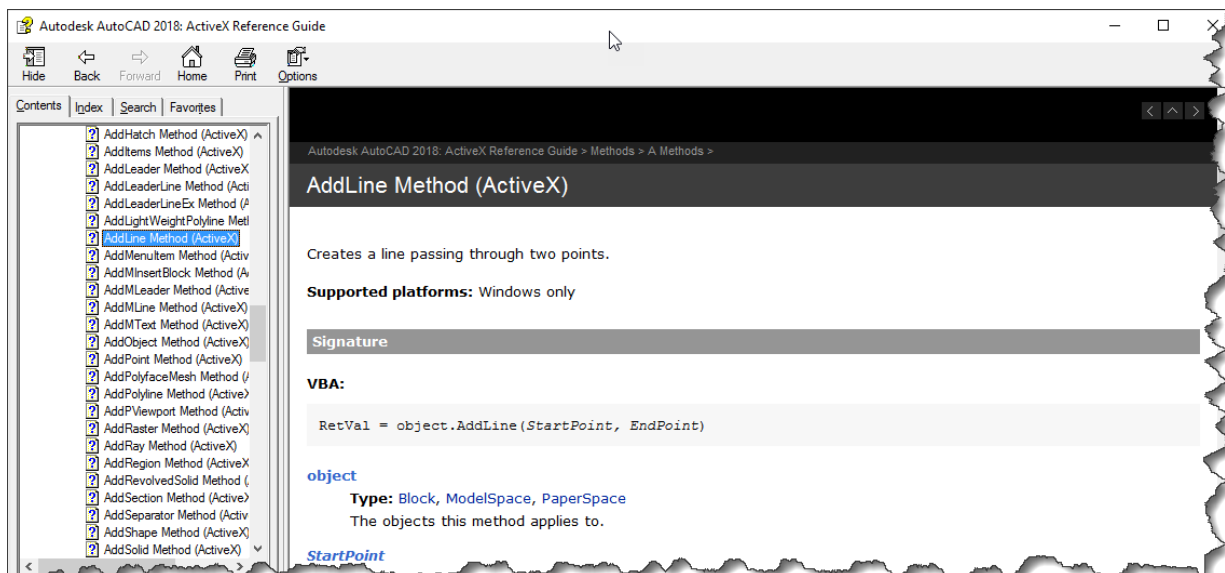
The *Layer* property is not Read-Only which means it can be set with a value indicating the new layer. But according to the format of how these functions are called, we need to specify the Line object first so that will be our first argument to the function.

The next thing is the value for the Layer name. This would then be the second argument to the function. Calling that function would look something like this... (***vla-put-layer lineobject layername***).



I know you're catching on to all this so this example includes error checking. It also includes an additional error checking function (*vlax-write-enabled-p*) that will tell you if the specified object is allowed to be modified or not. This would prevent an error if for some reason a particular object couldn't be modified. The only thing to watch here is that the Layer you are going to change to already exists in the drawing.

This covers extracting and setting properties quite well. The next thing we need to cover is Methods. For this example, we'll create a line using ActiveX in Visual LISP.



From the help for the *AddLine* method, you can see that Visual Basic requires two arguments. In addition, the *Object* lists several possibilities. The Modelspace collection, Paperspace collection and a Block object. That means that a Line needs to be in either Paperspace, Modelspace or part of a Block definition. In this case, we'll be adding it to the Modelspace collection. What this means

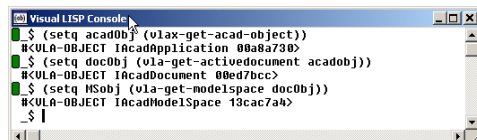
is that our (*vla-addline*) function will need three parameters, the Modelspace object being the first parameter, and Start Point as the second and the End Point as the third.

We'll start with the first parameter. This is the Modelspace collection object. You can't really just select the Modelspace object in AutoLISP so the (*vlax-ename->vla-object*) function that was used before doesn't really help in this case. For this task, you need to start from the root of the Object Model and that's the Application Object. To get the application object, you can use the (*vla~~x~~-get-acad-object*) function.

Now if you look at the *Application* Object from the Object Model in the help, you'll see there's an *ActiveDocument* property. This gives you the object of the currently open, active document in the *Documents* collection. With AutoCAD being able to have multiple documents open at once, each open document is stored in the Documents collection. You could iterate through all the different documents in the *documents* collection until you get to the one *document* with its *Active* property set to *True* but this *ActiveDocument* property is a shortcut through all that work. Now that you know how to get the application object, and how to get to the document level, you can figure out how to get the *document* object. You'll use the (*vla-get-activedocument*) function passing the Application object as its argument.

Finally, from the *Document* object, you need to get to the *ModelSpace* object. If you look at the *Document* object (not the *Documents* collection – remember we skipped past it with *ActiveDocument*), you'll see the Modelspace property. This property returns the *ModelSpace* collection which is what we need to tell Visual LISP where to add the line. To jump to the *ModelSpace* collection object, you'll be using the (*vla-get-modelspace*) function that returns the *Modelspace* object we need. The code we'll eventually end up using looks something like one of these two examples. You'll see one takes each, one step at a time (if you think you'll need the *Application* or *ActiveDocument* objects again) and the other does them all at once (if you won't need the *Application* or *ActiveDocument* objects again later).

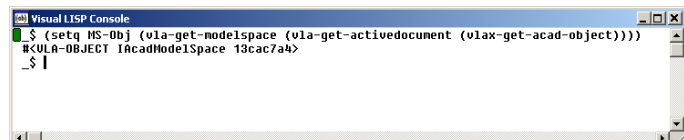
The next step is getting the points we need for the start and end of the line. Looking back at the *AddLine* method, you'll see it specifies the *StartPoint* and *EndPoint* as a *Variant* – 3 element Array of *Doubles*. A "*Double*" is a Visual Basic data type similar to a *Real* number. And *Array*, is Visual Basic's equivalent to a *List*. And finally, a *Variant*, is a special Visual Basic data type that can hold any data type (*string*, *real*, *double*, etc.).



```

$ (setq acadObj (vla-get-acad-object))
#<VLA-OBJECT IAcadApplication 00a8a730>
$ (setq docObj (vla-get-activedocument acadObj))
#<VLA-OBJECT IAcadDocument 00ed7bcc>
$ (setq MSobj (vla-get-modelspace docObj))
#<VLA-OBJECT IAcadModelSpace 13cac7a4>
$ |

```



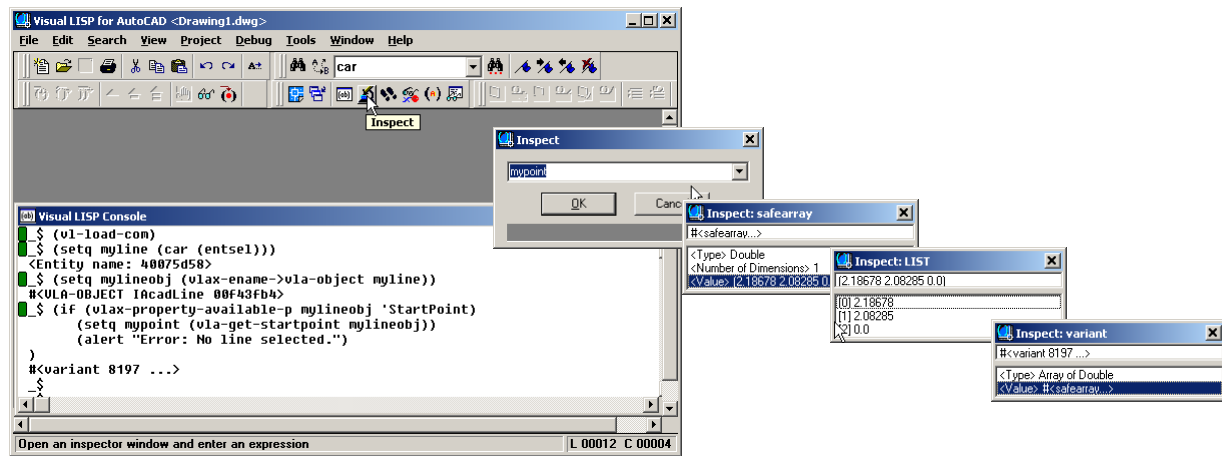
```

$ (setq MS-Obj (vla-get-modelspace (vla-get-activedocument (vla-get-acad-object))))
#<VLA-OBJECT IAcadModelSpace 13cac7a4>
$ |

```

This is a lot different than how AutoLISP specified points with a list of three *Real* numbers. It sounds complicated but once you get used to it, it's not a big deal. The best way to see how a point like this is structured is to extract a point from an existing line and look at how it's structured. This way, AutoCAD shows you what it uses, and then you can duplicate it. Let's look at the endpoint of the line we previously changed the layer of using the (*vla-get-startpoint*) function.

Now that you have the *Variant* as it was returned by the *StartPoint* property using the code shown in the image above, you can examine it further with the *Inspect* tool in the Visual LISP editor.



As you can see, by using the *Inspect* tool, and double clicking on the results, you can progressively work your way into the contents of the variable deeper and deeper to see exactly how it's constructed. You can also see that the end result is a list of three *Real* numbers. This is where we'll start to build our point. Next, this list needs to be built into a *Safe Array*. This can be done with the (*vlax-make-safearray*) function. This function is called with a couple arguments which are explained as follows...

(vlax-make-safearray type '(l-bound . u-bound) [(l-bound . u-bound)]...)

type = Constant or Integer indicating the type of the safe array

- vlax-vbInteger* (2)
- vlax-vbLong* (3)
- vlax-vbSingle* (4)
- vlax-vbDouble* (5)
- vlax-vbString* (8)
- vlax-vbObject* (9)
- vlax-vbBoolean* (11)
- vlax-vbVariant* (12)

l-bound = lower array limits (integer)

u-bound = upper array limits (integer)

Calling this function just initializes the array. It doesn't actually put any data values in it. To fill the array with values, you'll use the function (*vlax-safearray-fill*). This fills the data into the array that was just created. Once this is done, you can convert this safe array into a variant by using the (*vlax-make-variant*) function. Here's a description of (*vlax-make-variant*).

(vlax-make-variant [value] [type])

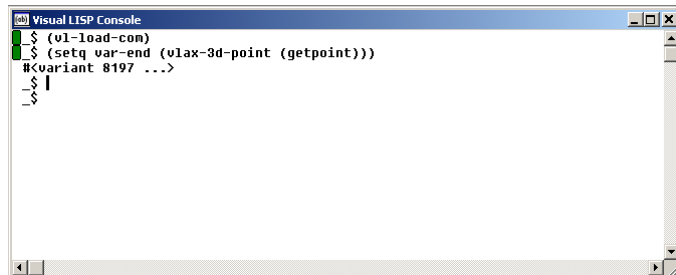
value = The value to be assigned to the variant

type = Constant or Integer indicating the type of variant

- vlax-vbEmpty* (0)
- vlax-vbNull* (1)
- vlax-vbInteger* (2)
- vlax-vbLong* (3)
- vlax-vbSingle* (4)
- vlax-vbDouble* (5)
- vlax-vbString* (8)
- vlax-vbObject* (9)
- vlax-vbBoolean* (11)
- vlax-vbArray* (8192)



After this last step, the point is now ready for use by Visual LISP's (vla-addline) function. Here's how it would look to create this point in the Visual LISP editor. Notice that when the function (vla-safearray-fill) is used, you don't need to save its return value although you can if you like.



```

Visual LISP Console
$ (vl-load-com)
$ (setq var-end (vlax-3d-point (getpoint)))
#<variant 8197 ...>
_$ |
_$

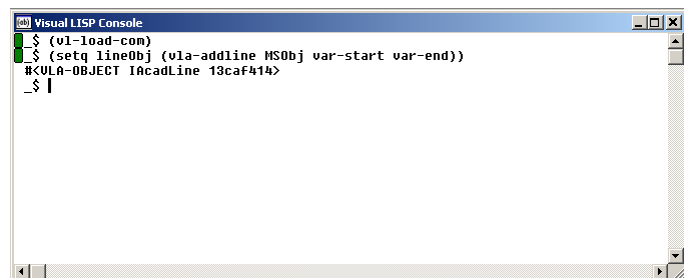
```

This is a lot of work just to get a point for use in Visual LISP ActiveX. You could always make a wrapper function to do this or, you could use the easier to use function (vlax-3D-point). This function takes a list, or separate coordinate data like you are used to dealing with in AutoLISP and returns a safe array. Let's take a look at how that's done for the line's EndPoint.

You can see from this image, that it's a lot easier with this function to create a 3d point. This is really the best way to go about it. The only reason I showed you the other method was because there are times when working in ActiveX from Visual LISP that you'll need to create an array in a similar fashion when it doesn't involve point coordinates.

So now that we have our ModelSpace collection object and our two points for the start and ending of the line, we're ready to use the (vla-addline) function to create a new line in AutoCAD. This is done by calling the function as we discussed earlier and looks something like this...

Don't forget to save the results of your (vla-addline) function to a variable. This function returns the created Line ActiveX object that can then be used if you want to set the Layer or Linetype or some other property of the newly created line.



```

Visual LISP Console
$ (vl-load-com)
$ (setq lineObj (vla-addline MSObj var-start var-end))
#<VLA-OBJECT lAcadLine 13caf414>
_$ |
_$

```

You now should have enough information to do just about anything in AutoCAD that its object model supports. You've extracted and set property values for existing geometry. You've also created geometry from scratch. Don't worry that all you did was change and create a line.

If you are familiar with the AutoCAD Object Model and follow the procedures demonstrated here, you'll have enough information to find out what the Visual LISP ActiveX function is named, and you'll be able to find out how to call it. Just remember that if a particular function requires an argument and you don't understand what format it's looking for, the easiest thing to do is to create the same thing manually in AutoCAD. Using Visual LISP and its editor, you can then extract the existing value of what you created manually to figure out what you need to do in your code. It's a simple technique but it'll have you lots of headaches in the end.