VisualAPL Tutorial

This section contains a complete, start-to-finish tutorial on developing with the VisualAPL programming language. It discusses a wide range of language features, as well as intrinsic .NET objects, commonly used objects for daily programming tasks. Of course, there is always more to learn, so keep checking the VisualAPL Forum for updated information.

The VisualAPL development team wants to hear from you, so use the VisualAPL Forum to report your experiences, described your suggestions, highlight bugs, tell others about your successes and enhance the VisualAPL community with your thoughts.

1 What is VisualAPL?

VisualAPL is a next generation APL. Specifically it is APL without limits and designed for the .Net framework. The .Net framework can be looked at the ultimate batteries-included programming enviroment, with built-in tools to accomplish almost every computing task.

VisualAPL is a .Net language and a peer of C#, VB and managed C++.

VisualAPL is integrated with Visual Studio.

VisualAPL contains the rich operator set of APL, the right to left execution, the ability to create user-defined functions, dynamically-typed variables, and dynamic code execution.

However, VisualAPL is much more. VisualAPL also implements much of the C# language syntax. This means that with VisualAPL you can build any application, access any resource directly, and are never limited by the language or the language's access to resources. Never again wait for the next release of APL to have access to the latest windows controls or OS features. Best of all, the enhanced syntax is based on the C# ECMA standard.

VisualAPL is also object-oriented, with all the features that object-orientation brings to a language.

VisualAPL is fully-dynamic with interactive interpretation of source code just like all APL's.

VisualAPL also has static compilation which produces dll's and exe's which can be called from other .Net languages such as C#, VB, managed C++ and many more.

Assemblies created with VisualAPL are verifiable and managed. There are no native libraries referenced or used, and all of the assemblies created are 100% .Net.

2 What are the differences with legacy APL's?

VisualAPL is APL for .Net. It is not a replication of any particular APL or APL environment. The principle development environment is Visual Studio. This means that development of VisualAPL is integrated with the development of all other .Net languages, such as C#, VB, and managed C++. However, this new development environment brings new features and concepts when creating APL applications. Because of this integration, VisualAPL projects integrate seamlessly with other .Net language projects both during development, working in the same solution, during debugging and at run time.

Moving existing APL code to VisualAPL is trivial. VisualAPL uses the Unicode standards for all APL characters, supports APL syntax and provides "Paste APL+Win" to facilitate importing legacy code. VisualAPL can also create objects which are COM compatible. This means that VisualAPL assemblies can be used in legacy APLs as a COM objects. So as you move your legacy APL code to VisualAPL, you can use your new VisualAPL .Net assemblies (dll's) from your legacy APL applications.

The most significant difference between VisualAPL and legacy APLs is in the use of objects and scoping.

Like legacy APLs, VisualAPL has both local and global variables. However, legacy APLs have dynamic localization in which names localized in a function definition were known to further functions invoked within it. According to Ken Iverson:

"This decision made it possible to pass any number of parameters to subordinate functions, and therefore circumvented the limitation of at most two explicit arguments, but it did lead to a sometimes confusing profusion of names localized at various levels. The introduction of atomic representation (box and enclose) has made it convenient to pass any number of parameters as explicit arguments; in J this has been exploited to allow a return to a simpler localization scheme in which any name is either strictly local or strictly global."

VisualAPL follows the same simpler localization scheme preferred by Ken Iverson, "in which any name is either strictly local or strictly global".

In addition, this convention brings VisualAPL into compliance with all other .Net languages and makes the integration with those languages transparent when using development systems such as Visual Studio.

It is also possible to reasonably simulate the legacy dynamic localization using classes, and examples of this have been created.

In VisualAPL a variable is local to the function in which it is created unless specified as global. VisualAPL also supports the C# function syntax in addition to the traditional APL function syntax. This means that passing variables between functions is both clear and not limited to only two variables. A function can have an arbitrary number of named arguments, with defaults if desired, so the legacy dynamic localization is no longer necessary.

Because legacy APLs are based on a native interpreter, they are not and can not be managed or verifiable. This also means that APL functions which used native code, such as assembler routines, for activities such as TEXTREPL and WHERE, are also not manageable or verifiable . These same facilities are available in managed code, but the old assembler routines, can not be part of the managed and verifiable environment.

Objects and object oriented programming will also be a new concept for many APL programmers. At first the idea of objects may be a bit confusing, but with a bit of use, they will become an indispensable part of development.

Perhaps the easiest way to think about an object is to consider it a box. You can open the box and add things to the box, change the contents of the box, but the box does not change, just it's contents. Of course, you can also replace the box with a new box. As you use VisualAPL you will find this object-oriented approach both extremely useful and maybe irritating at first.

Most importantly, by having an object-oriented approach, you can now do anything any other language can

do, access any resource, create any object any one else can, and most importantly, it makes you a first class citizen of the Windows programming world.

VisualAPL also uses the same set of keywords that are used by C# and most other programming languages.

3 Is VisualAPL fast?

The simple answer is that VisualAPL is just as fast or faster than any .Net language. VisualAPL will even outperform C# in most cases. However, because VisualAPL has a rich and robust syntax, you will discover that performance can vary depending on how a solution is created.

The examples in this tutorial cover a number of methodologies for both writing code quickly and also creating the a solution with the fastest possible performance.

Since the .Net framework represents a major improvement over the legacy Win32 API, most applications will perform better in the .Net environment.

VisualAPL also supports adding "strong data typing" to your functions to make them perform as fast as possible. Using Visual Studio code performance analysis tools can identify the functions which might need strong data typing to improve their performance.

4 Compatibility with .Net and other .Net langauges?

The .Net world is based on a scheme of top level objects called types. The primary top level type you will create is a class. Classes are very much like a workspace, in that they contain both methods (functions) and fields (global variables), but they can also contain a variety of other objects, as well as other types.

To interact with the .Net world, these top level classes follow a formal naming convention.

This naming convention is designed to provide the developer with the ability to create a kind of hierarchy of classes and bundle classes which are for a common purpose under a common name.

This grouping of classes is called a namespace.

For instance, if you were creating a class which did amortizations and a class which did financial plotting you might want to place them together in a namespace you would name finance. So the names would be:

- finance.amortizations
- finance.finplot

As you can see, this convention is the name of the namespace followed by a dot and then the name of the class.

We have followed this naming convention in VisualAPL, so that once you have created your .Net project, any other .Net language can load and use your application.

5 What about workspaces?

Legacy APL workspaces have always used a proprietary internal format, so they would only work with a particular APL interpreter. As a .Net language VisualAPL source code is maintained in Unicode text files suitable for any text editor. This is one of the principals of .Net and Visual Studio. Everything is open and readable, nothing is proprietary. Protecting your source code is easy too, because your production-ready .Net assemblies are compiled.

6 What does the source file look like?

The basic construct of a VisualAPL .Net assembly source code looks something like this:

```
using System

namespace mynamespace {

      public class myclass {

            ⍝ this is a global variable, and is referred
            ⍝ to as a field in .Net languages
            ga ← 100 200 300

            ⍝ these are user defined functions
            ∇r←a add b {
                  r←a+b
            }

            ∇r←a minus b {
                  r←a-b
            }

            ∇r←conjugate b {
                  r←+b
            }

            ∇r←getga {
                  r←ga
            }
      }
}
```

That is all there is to a basic APL workspace in a .Net text file. Note that a class is roughly equivalent to a workspace.

The class would be referenced as mynamespace.myclass from other classes.

The using reference at the top of the sample indicates that the class you are creating will rely on the functionality found in the .Net System namespace. You might also have:

```
using System.Windows.Forms
```

in the event you were building an application which used windows forms.

The reason the using statements are needed is that each dll or exe is only granted access to those resources it specifically requests when created or used as part of the managed and verifiable framework.

7 Is there an Interpreter?

Yes, VisualAPL still provides code interpretation on-the-fly.

You can still ⎕def a function from its text representation while you are running code.

You can use the APL 'execute' operator to run code like: ⍎"a←1+1".

However, the code you create, when run on a particular machine goes through the .Net Just-In-Time compiler to optimize performance for the machine on which the code is being executed.

8 VisualAPL Tutorial

This tutorial is an informal introduction to VisualAPL. It is not meant to replicate either the information you will find in your APL manuals or Microsoft's documentation of C# and .Net. While both APL and C# syntax should apply in virtually all cases, this tutorial provides the basic structure of syntax differences are reconciled.

As VisualAPL is a .Net language, all of the .Net namespaces, types and other objects are available for use.

In fact, you will discover that you can use almost all of your legacy APL code as well as the vast majority of C# examples available on the Web and in the MSDN. Care should be taken to check C# examples which rely on precedence, such as multiply before addition. In these cases, you may need to add parenthesis, as VisualAPL executes within any statement from right to left.

However, our experience is that, in most cases, the C# examples already use parenthesis to establish this precedence.

9 General Overview

In the following sections, you will find a general overview of the programming practices and patterns when using the VisualAPL programming language.

10 Using VisualAPL as a desktop calculator

To display your immediate-mode, VisualAPL session in Visual Studio, use

"View Menu > Other Windows >Other Windows > Cielo Explorer"

The Cielo Explorer session is independent from any project or solution you are currently editing or viewing in Visual Studio. The Cielo Explorer session should appear very much like any APL session you may have previouslyencountered. For instance typing the following expression and then the Enter key will illustrate the calculated result:

```
    1 2 3 + 1 2 3
2 4 6
```

You can make use of both .Net types you have referenced and also other dll's. The .Net framework is organized into namespaces based on the types you will need for your project. There are over 4,000 types in the .Net framework. At the root of these types is System. So executing the line:

```
using System
```

in your session, will assure that you have access to all of the intrinsic types in .Net, such as Int32, Int64, UInt64, UInt64, Single, Double, String, Byte, Char, etc.

It is extremely important that VisualAPL support all types created in .Net, without this it would be impossible to integrate with other .Net programs.

For instance, when you type:

```
  a  = "hello"
```

by default a is a string type. It will work the way you expect with the APL operators, but it is a .Net String object.

To see how this works, try this:

```
   a.IndexOf("el")
1
```

you should see a 1 returned.

Methods created on objects from other .Net languages are based on an []io of 0. Obviously setting ⎕io in

your session will not affect the way the methods which belong to other objects work, but it will affect the APL operators you have defined in your VisualAPL .Net assembly.

11 APL Operators and Functions

In general the APL operators and functions included in VisualAPL are designed to be compatible with IBM APL2. There are a few exceptions based on object requirements and language compatibility.

One of those exceptions if the use of the equal (=) symbol.

The APL equals has always actually been an approximate equals, based on []ct. The APL equal symbol has always been accessed using the "Alt+5" key stroke. In the operator set used by VisualAPL the Unicode APL symbol for approximately equal (≈) is used for APL equality comparison.

As in all other .Net languages, in VisualAPL = operator means "assign by reference".

The VisualAPL ← assignment operator means "assign by value", the same as in legacy APLs.

Having the new = assign by reference operator provides some powerful options. Remembering that everything is an object in VisualAPL, you can try the following:

```
  a ← 1 2 3 4 5
      b = a
      a
1 2 3 4 5
      a[1] = 500
      a
1 500 3 4 5
      b
1 500 3 4 5
```

This works because we have put new data in the object, but not reassigned the object. For instance, if we do this:

```
  a←a

      a[1] = 20
      a
1 20 3 4 5
      b
1 500 3 4 5
```

Then the reference is broken, and further index assignments to either a or b will not affect the other object.

12 Strings

All strings or character arrays are Unicode by definition. For compatibility with .Net, the backslash (\) is the escape character in string parsing. For instance:

```
   'hello'
hello
      'doesn\'t'
doesn't
      "doesn't"
doesn't
      '"Hello", she said.'
"Hello", she said.
      "\"Hello\", she said."
"Hello", she said.
      '"Don\'t", he asked?'
"Don't", he asked?
```

Create the following function in a Cielo Explorer script to see implied line continuation for string literals.

```
▿makestring {
      a = "this is a
      line of text
      over three lines"
      □←a
      a = "this is a\n
      line of text\n
      over three lines"
      □←a
      a = @"another line
      of text over
      several lines"
      □←a
      a = @"another line\n
      of text over\n
      several lines"
      □←a
}
```

Notice that those string literals which start with the @ symbol are interpreted raw, with escape characters included. Note also that white space is preserved in the raw strings.

For compatiblility you can also add strings:

```
a = "hello, "
      b = "goodbye"
      a+b
hello, goodbye

      a = "\\\u0066\n"
```

a now contains backslash, letter f, new line.

Note
The escape code \udddd (where dddd is a four-digit number) represents the Unicode character U+dddd.

The advantage of @ quoting is that it is simple to create fully qualified file names

@"c:\Docs\Source\a.txt" rather than "c:\\Docs\\Source\\a.txt"

To include a double quotation mark in an @-quoted string, double it:

```
@"""Hello!"" he yelled."
"Hello!" he yelled.


As with all .Net types, strings have a wealth of built in methods.  For
instance:
      a = "hello"
      a.Length
5
      a.Substring(2)
llo
      a = "    hello    "
      a.Length
13
      a.Trim()
hello
      a.Trim().Length
5
      b = a.Trim()
      ρb
5

The same is true for integers, doubles, etc.  For instance:
      a = 10
      a.MaxValue
2147483647
      a.MinValue
-2147483648
```

As well as the intrinsic types, there also new collection types. These .Net types make is trivial to create data which can be consumed by any .Net language.

To have access to the .Net collection datatypes, include "using System.Collections" in your script or assembly.

13 Using System.Collections

The .Net framework includes many new data types. The Collections namespace contains types which are useful for creating data types which can dynamically add, remove and manage elements. Creating an array can be done using an ArrayList.

Before this will work, add "using System.Collections" to either your project or execute it in the session.

```
  a = ArrayList()
      a.Add(10)
      a.Add(100)
      a.Count
2
      a.GetType()

System.Collections.ArrayList
```

Another powerful collection is the Hashtable:

```
   a = Hashtable()
      a["test"] = 100 200 300
      a["hello"] = "some text"
      a["test"]
100 200 300
      a["hello"]
some text
```

The newest addition to .Net 2.0, Generics, are also supported. To include generics in your project, place this at the top of your file, or run it in the session:

```
using System.Collections.Generic
```

Generics are collections which can be typed. This provides both an increase in speed and compatibility with other .Net programs. For instance, the Generic Dictionary is similar to the Hashtable shown above, however it is instantiated in a different manner:

```
  a = Dictionary[String, Int32]()

      a
System.Collections.Generic.Dictionary`2[System.String,System.Int32]

      a.Add("more", 10)


      a["more"]

10


      a.Add(10, "more")
```

```
bad args for method
```

As you can see, only keys of string type and data of integer type can be added to the collection.

You can use ⎕wi in the session as well, assuming the appropriate namespaces are referenced:

```
   "fm" ⎕wi "Create" "Form"

fm

      "fm.b" ⎕wi "Create" "Button"

fm.b

      "fm.b" ⎕wi "where" 10 10
      "fm.b" ⎕wi "caption" "Click"

      ƒ push(a,b) {⎕←a;⎕←b}

      "fm.b" ⎕wi "onClick" "push"

      "fm" ⎕wi "Show"
```

You can also use the .Net System.Windows.Forms directly by including this reference at the top of the file:

```
refbyname System.Windows.Forms
using System.Windows.Forms
```

You can also just enter these two lines in the session. After these .Net assemblies are available to your session, the following will create a form:

```
  a = Form()

      b = Button()

      a.Controls.Add(b)

      b.Text = "Click"

      ƒ push(a,b) {⎕←a;⎕←b}

      b.Click += push

      a.Show()
```

If we push the button the following is displayed to the session:

```
System.Windows.Forms.Button, Text: Click

System.Windows.Forms.MouseEventArgs
```

In the above examples VisualAPL functions were defined using the "Alt+f" keyboard shortcut. The VisualAPL function signature will be described in more detail in a subsequent sections of the tutorial.

14 Controlling Program Flow

Each statement in VisualAPL is executed from right to left. Separate statements in VisualAPL are executed in sequential order, unless that sequential order is modified by "Program Flow Control" statements.

Using the conditional if statement
Either the APL syntax and the C# syntax may be used in VisualAPL.

```
:if x < 10
      a←100
:else
      a←200
:endif
```

alternatively:

```
if (x < 10) {
      a←100
} else {
      a←200
}
```

In addition, multiple else if statements can be included:

```
:if x < 10
      a←100
:elseif x > 100
      a←200
:else
      a←300
:endif
```

alternatively:

```
if (x < 10) {
      a←100
} else if (x > 100) {
      a←200
} else {
      a←300
}
```

The last "else" in both cases is optional.

In addition, you can use the then/else control structure:

```
a = (x < 10) then x+100 else x+200
```

This following statement syntax is also valid:

```
   a = (x < 10) then (x > 3) then x+100 else x+200 else x+300
      a
105
```

Using for loops:
Again, both the classic APL and C# "for" loop syntax is supported:

```
:for x :in ⍳10
      a←x+1
:endfor

foreach (x in ⍳10) {
      a←x+1
}
```

In addition, the for loop with counter is supported:

```
for (i = 0;i<10;i++) {
      a←i+100
}
```

Notice that i++ and i-- are also supported, this works for both scalars and arrays.

The "for" loop with "to" and "step" is also supported:

To quickly iterate 100 times:

```
for (1 to 100) {
      a←x+y
}
or set the step and counter variable:
for (i = 1 to 100 step 2) {
      a ←i+100
}
```

This "for" loop localizes the counter variable to the loop, so that:

```
i = 22.3
for (i = 1 to 10) {
      ⎕←i
      for (i = 10 to 30) {
            ⎕←i
      }
}
⎕←i
```

This will display the i value for the outer for and then the i values for the inner for and finally display:

```
22.3
```

The for loops with counter also support an else control:

```
x←0
for (i = 0;i<x;i++) {
      ⎕←i
} else {
      ⎕←"here"
}
```

The above code will display in the session as follows:

```
here
```

If the for loop is never entered then the else runs.

Using while loops:

Both the APL and C# while loop syntax is supported:

```
:while x < 10
      x++
:endwhile

while (x < 10) {
      x++
}
```

In the case of the while loop, you can also use the else control:

```
while (x < 10) {
      x++
} else {
      x←100
}
```

If the while loop is never entered then the else runs.

do...while or :repeat...:until loop

```
:repeat
      x++
:until x >= 10

do {
      x++
} while (x<10)
```

break, continue and :continue and :leave statements in loops

break and :leave both exit the immediately enclosing loop :continue and continue statements, continue with the next iteration of the loop

switch and :select control flow

```
:select choice
      :case b
            a←100
      :case c
            a←200
      :caselist d e
            a←300
      :else
            a←1000
:endselect

switch (choice) {
      case b:
```

```
            a←100
            break
      case c:
            a←200
            break
      case d:
      case e:
            a←300
            break
      default:
            a←1000
            break
}
```

The break is required in each case statement in the switch.

All comparisons for selection are done using the identity operator.

goto and :goto statement

```
goto L1
and
:goto L1
both branch to a label L1:
```

However, in VisualAPL it is not possible to branch to a line number or select labels from an array.

Labels must be a specific label destination.

15 Defining Functions

User defined functions are created as follows:

```
∇r←a fn2 b {
      r←a+b
}
∇r←fn1 b {
      r←b
}
∇r←fn {
      r←100
}
∇a fn2 b {
      a+b
      ⎕←a+b
}
```

Except for the use of the {} to begin and end the user defined function the syntax is identical to classic APL function definition. However, statements that return a value do not display when run in a function unless they are explicitly output, in this case using the ⎕←a+b

In addition it is not necessary to always assign the return variable, for instance:

```
∇r←a fn2 b {
      return a+b
}
```

will return the value of a+b even though r was not set.

Checking for the left argument has also been enhanced with ⎕monadic or ⎕dyadic:

```
∇r←a fn2 b {
      :if ⎕monadic
            r←b
      :else
            r←a+b
      :endif
}
```

In addition to the classic APL user defined functions, VisualAPL also supports function signatures compatible with all .Net languages. This is especially important when you are creating a class which may be consumed by another .Net language such as C#. It is also important when you are consuming a method on a class created by another .Net language. For instance:

```
function fn2(a, b) {
      return a+b
}
```

VisualAPL provides a new function definition, using the "Alt+f" keystroke which displays the glyph of the mathematical function symbol: f

```
f fn2(a, b) {
      return a+b
}
```

```
or
∫ r←fn2(a, b) {
      r←a+b
}
or
∫ fn4(a, b, c, d) {
      e = a+b
      e = e×c+d
      return e
}
```

Since everything is an object in VisualAPL, you can assign a function to a variable at creation or even later:

```
myfn = ∫ fn3(a, b, c) {
      return a+b+c
}
```

Then you could run:

```
   fn3(1, 2, 3)
6
```

or you could run:

```
   myfn(1, 2, 3)
6
```

You could also place this in an array:

```
   myarr = myfn myfn myfn
      myarr[1](1, 2, 3)
6
```

You could also assign the function to a variable this way:

```
   myvar = fn3
      myvar(1, 2, 3)
6
```

16 More about defining functions

Methods can also have default arguments:

```
function fndef(a, b = 10, c = "hello", d = myfn(1, 2, 3)) {
      ⎕←"a" a
      ⎕←"b" b
      ⎕←"c" c
      ⎕←"d" d
}
```

This function can be called with from one to four arguments:

```
   fndef(100)
      fndef(100, 200)
      fndef(100, 200, 300)
      fndef(100, 200, 300, 400)
```

will all work equally well. When an argument is missing, the value for the argument is set to the default.

You can also call this function with the arguments rearranged by using their names:

```
   fndef(b = 400.3, a = 99)
```

However, you must always set the value of a, either by position or name, as it does not have a default.

Or you can call this function by order, leaving out values:

```
   fndef(10,20,,500)
```

In this case the value of c is the default "hello"

In addition, you can pass an argument list to a function using the ⎕arglist keyword:

```
   args = 10 20 30 40
      fndef(⎕arglist args)
```

You can also create a matrix of named arguments and values and pass them using ⎕argnames:

```
   args = 5 3⍴"a" 100 "desc1" "b" 200 "desc2" "c" 300 "desc3" "d" 400 "desc4"
"x" 500 "desc5"
      fndef(⎕argnames args)
```

Notice that the arguments a,b,c and d will be set to 100, 200, 300 and 400. This provides the ability to call a function with a matrix of potential arguments and have it select only that that apply to it. Also notice that you can have more columns than just the name and value columns. This makes it possible to include argument descriptions or alternate values in addition columns.

When combining position arguments and named arguments, there can not be positioned arguments after named arguments, for instance:

```
fndef(10, c = 99, 100)
```

Would be illegal.

You can also combing ⎕arglist and ⎕argnames:

```
   argsp = 10 20
      argsn = 2 2⍴"c" 88 "d" 99
      fndef(⎕arglist argsp, ⎕argnames argsn)
```

Important Feature: Default values for parameters are evaluated only once. This is an extremely powerful feature, but can also be disconcerting if misunderstood.

For instance if we have the following function definition:

```
public ƒ outerfn(a) {
      return ƒ innerfn(b = a) {
            return b
      }
}

      c = outerfn(10)
      c()
10
      c = outerfn(100)
      c()
100
```

However, if we use an object, such as the ArrayList collection as our default value, then values accumulate in the ArrayList.

```
function fn(a, al = ArrayList()) {
      al.Add(a)
      return al
}
      a = fn(10)
      a.Count
1
      a = fn(20)
      a.Count
2
      a = fn(30)
      a.Count
3
      foreach (n in a) {⎕←n}
10
20
30
```

Each call to the function adds information to the instance of the ArrayList which was assigned to al when the function was instanced.

If you want the argument to default to an ArrayList but not accumulate data, this construct will work:

```
function fn(a , al = null) {
      if (al == null) {
            al = ArrayList()
      }
      al.Add(a)
      return al.Count
```

```
}
      fn(10)
1
      fn(20)
1
      fn(30)
1
```

In this case there is no accumulation of argument data.

One of the features of many new languages are code blocks or closures. These are created within a function and can be called at any time with the variables set to the values when the reference to the closure or code block was created.

To accomplish this, we allow for the creation of anonymous functions as well as named functions.

In its simplest form:

```
function outerfn(a, b) {
      c ← a×b
      return ƒ (first = a, second = b, third = c) {
            □←first
            □←second
            □←third
      }
}
```

In this case we return a pointer to the closure with the arguments preset to the variables in the function where the instance is created:

```
   p = outerfn(10, 20)
      p()
10
20
200
      p = outerfn(30, 40)
      p()
30
40
1200
```

17 Typing Arguments to Functions

It is also possible to specify the data type of an argument, this is particularly useful when a function is going to be consumed by another language, such as C#. Functions which include typed arguments and returns must be defined in classes and cannot be defined directly in the session.

```
function mytfn(int a, string b) {
      □←a
      □←b
}
```

This function can only be called with an integer first argument and a string second argument, and the function signature will require this when it is called from another language. You can call it like this:

```
   x1 = 100
      x2 = "hello"
      mytfn(x1, x2)
100
hello
```

You can also specify the return type:

```
function Int32 mytfn(Int32 a, Int32 b) {
      return a+b
}
```

This function requires two integers and returns an integer. Again, when this function is consumed by other languages they will see that integer arguments are required and are assured that only an integer will be returned. This provides both speed and verifiability.

When you create a function, it is only available to your assembly by default. If you want other programs to be able to consume it, you will need to explicitly make it public:

```
public function Int32 mytfn(Int32 a, Int32 b) {
      return a+b
}
```

This function can now be seen by other languages who use your class.

When you specify a return type, you can leave out the keyword function:

```
public Int32 mytfn(Int32 a, Int32 b) {
      return a+b
}
```

When you want to create a function which does not return a value, the void keyword is used:

```
public void mytfn(Int32 a, Int32 b) {
      c = a+b
}
```

Other languages who use this function will see that it has no return value.

It is also possible to pass arguments by reference, using ref and other modifiers with function arguments require that the function be defined within a class and will not work when defining functions dynamically in the session.

```
function myref(a, b, ref c) {
      c ← a+b
}

      a = 100
      b = 200
      c = 99
      myref(a,b, ref c)
      c
300
```

It is also possible to pass an arbitrary number of arguments to a function:

```
function myarb(a, b, params c) {
      ⍞←"a" a
      ⍞←"b" b
      foreach (d in c) {
            ⍞←"c" d
      }
}
```

This function will take an arbitrary list of arguments, with the first two being a and b and the rest placed in c

```
   myarg(1, 2, 3, 4, 5)
a 1
b 2
c 3
c 4
c 5
```

The params option can also be the only argument

```
function myarb(params a) {
      foreach (d in a) {
            ⍞←"a" d
      }
}
```

Now myarb can be called with any arbitrary number of arguments from 0 to ....

One of the primary advantages of params is that other languages, such as C#, can pass an arbitrary number of arguments to your function. The same is true for the ref modifier, which allows languages such as C# to pass an argument to your function by reference.

18 Data Types and Collections

The .Net framework includes numerous types for handling data.

There are the intrinsic types, such as long, float, double, int, etc.

There are also collections and generic collections. The importance of these data structures is both their usability and their common availability to all .Net languages. Making passing data between applications simple and efficient.

The ArrayList: a Heterogeneous, Self-Redimensioning Array

This is similar to an APL heterogeneous array. Creating the ArrayList is as simple as:

```
a = ArrayList()
```

You can add any type of data to the arraylist:

```
a.Add(10)
a.Add(10 20 30)
a.Add("hello" 100 "something else" 99.4)
a.Add(3 3⍴⍳9)
```

ArrayLists can be used in foreach loops:

```
:for b :in a
      ⎕←b
:endfor
```

It is possible to access the data in an ArrayList in any order, for instance:

```
a[1]
a[2]
a[0]
etc...
```

a.Count returns the number of elements in the ArrayList. Microsoft provides excruciatingly detailed information on all of these data structures, and their use.

The System.Collections.Queue Class

The Queue class provides adding and removing items on a first come, first served basis.

The Queue class has an internal circular object array and two values that mark the beginning and ending of the array.

The Enqueue() method returns the current item from the head index. The head index item is set to null and the head is incremented. If you want to just look, use the Peek() method.

Most importantly, the Queue data structure does not allow the random retrieval of an item, as the ArrayList did. For instance you can not retrieve the second item in the queue without first dequeing the first item. However, there is a Contains() method which can be used to determine if an item is in the queue. The Queue is ideal for processing items in a specific order when it is needed by an application.

The Queue class implements a first in, first out or FIFO method of processing items.

The Stack class or First Come, Last Served

The Stack data structure makes it possible to access items on first in, last out order. Similar to the Queue class, the Stack class maintains items in a circular array. Data is exposed through two methods, Push(item) which adds an item to the stack, and Pop(), which removes and returns the item at the top of the stack.

The System.Collections.Hashtable Class

The Hashtable provides the ability to store data randomly using keys. When you provide a unique key, a new item is added. If the key exists in the table, the value is replaced.

The Hashtable has an Add method for adding items:

```
a = Hashtable()
a.Add("hello", "good morning")
a.Add("what", "is that")
a.Add("nums", (1 2 3 4 5))
```

To retrieve the data you can use simple indexing:

```
   a["hello"]
good morning
      a.ContainsKey("test")
false
      a["test"] = 100
      a.ContainsKey("test")
true
```

To find out is a key is contained in the Hashtable, you can use the ContainsKey method.

All of the classes in the System.Collection are available to use and work as documented in the Microsoft help files.

19 The Generic Collection

Generics provide the ability to restrict the types of data that can be added to a data structure. These data types are included in the .Net framework and are very useful for exchanging arbitrary data sized objects between languages.

List

There List data structure works like an ArrayList, but allows you to determine the type of data that can be added to the List. For instance:

```
a = List[string]()
```

This will create a List to which you can only add string data.

```
  a.Add("hello")
      a.Add("some more stuff")
      a.Add(10)
bad args for method
```

SortedList

If you ever wanted to have a sorted list with only unique keys, then the SortedList is perfect:

```
   a = SortedList[String, Int32]()
      a.Add("test", 100)
      a.Add("my info", 200)
      a.Add("abc", 300)

      foreach (n in a) {□←n}

[abc, 300]

[my, 200]

[test, 100]

      a.Add("abc", 300)
An entry with the same key already exists.
```

Dictionary

Represents a collection of keys and values, which is typed.

```
   a = Dictionary[String, Int32]()

      a.Add("one", 10)

      a.Add("two", 100)

      a.Add("three", 1000)
```

```
      foreach (n in a) {□←n}
[one, 10]
[two, 100]
[three, 1000]
```

SortedDictionary

The SortedDictionary generic class is a binary search tree with O(log n) retrieval, where n is the number of elements in the dictionary. In this respect, it is similar to the SortedList generic class. The two classes have similar object models, and both have O(log n) retrieval. Where the two classes differ is in memory use and speed of insertion and removal:

SortedList uses less memory than SortedDictionary.

SortedDictionary has faster insertion and removal operations for unsorted data: O(log n) as opposed to O(n) for SortedList.

If the list is populated all at once from sorted data, SortedList is faster than SortedDictionary.

```
   a = SortedDictionary[String, Int32]()
      a.Add("test", 100)
      a.Add("my", 200)
      a.Add("abc", 300)
      foreach (n in a) {□←n}
[abc, 300]
[my, 200]
[test, 100]
      a.Add("abc", 400)
An entry with the same key already exists.
```

Queue

The Queue is a first in, first out data structure which is typed.

```
   a = Queue[string]()
      a.Enqueue("one")
      a.Enqueue("two")
      a.Enqueue("three")
```

```
      a.Enqueue("four")
      a.Dequeue()
one
      a.Dequeue()
two
      a.Dequeue()
three
      a.Count
1
```

LinkedList

The LinkedList represents a doubly linked list.

```
   words = "the" "fox" "jumped" "over" "the" "dog"
      sentence = LinkedList[string](words)

      text = ""
      foreach (word in sentence) {text=text+" "+word}
      text
 the fox jumped over the dog
      sentence.Contains("jumped")
True
      sentence.AddFirst("today")
System.Collections.Generic.LinkedListNode`1[System.String]
      text = ""
      foreach (word in sentence) {text=text+" "+word}
      text
today the fox jumped over the dog
      mark1 = sentence.First
      sentence.RemoveFirst()
      sentence.AddLast(mark1)
      text = ""
     foreach (word in sentence) {text=text+" "+word}
```

```
      text
the fox jumped over the dog today
      sentence.RemoveLast()
      sentence.AddLast("yesterday")
System.Collections.Generic.LinkedListNode`1[System.String]
      text = ""
     foreach (word in sentence) {text=text+" "+word}
      text
the fox jumped over the dog yesterday
      mark1 = sentence.Last
      sentence.RemoveLast()
      sentence.AddFirst(mark1)
      text = ""
     foreach (word in sentence) {text=text+" "+word}
      text
 yesterday the fox jumped over the dog
      sentence.RemoveFirst()

current = sentence.FindLast("the")
      sentence.AddAfter(current, "old")
System.Collections.Generic.LinkedListNode`1[System.String]
      sentence.AddAfter(current, "lazy")
System.Collections.Generic.LinkedListNode`1[System.String]
      text = ""
     foreach (word in sentence) {text=text+" "+word}
      text
the fox jumped over the lazy old dog
      current = sentence.Find("fox)

sentence.AddBefore(current, "quick")
System.Collections.Generic.LinkedListNode`1[System.String]
      sentence.AddBefore(current, "brown")
System.Collections.Generic.LinkedListNode`1[System.String]
```

```
      text = ""

      foreach (word in sentence) {text=text+" "+word}

      text

 the quick brown fox jumped over the lazy old dog
```

Stack

The Stack generic data structure represents a variable size last-in-first-out (LIFO) collection of instances of the same arbitrary type.

```
   a = Stack[String]()
      a.Push("one")

      a.Push("two")

      a.Push("three")

      a.Pop()

three

      a.Pop()

two

      a.Count

1

      a.Peek()

one
```

20 Conditions for Flow Control

The structures **for**, **while** and **if** can contain conditions with any valid expression, the only requirement is that the expression return either true, false, 1, or 0.

```
a < b ≈ c
```

would evaluate b ≈ c first, then a < the result of b ≈ c

Two new operators are also supported, these are && and ||

In the case of && the expression to the left of the && is evaluated first, and if it returns a true or 1, then the right expression is evaluated. If a false or 0 is returned, then the right expression is never evaluated.

For instance:

```
a < b && b < c
```

only the a < b will be evaluated if a is greater than b

The inverse is true for ||, if the left expression returns a true or 1, then the right expression is never evaluated. If a false or 0 is returned, then the right expression is evaluated.

21 Comparing objects

When you are using objects, such as an ArrayList, it is important to remember that when checking to see if two objects are equal, the equal evaluation is only to see if the objects are referenced by the same pointer, not to determine if there contents are the same.

22 Error Handling

Errors are handled using the try, catch, finally flow control structure

For instance:

```
try {
      a←⍳9
      ind←12
      a[ind]
} catch {
      ⎕←"an error occured"
}
```

If an error occurs or is thrown within a try block of code flow control is passed to the try handlers, or catch.

The try handlers, or catch can be created either with arguments or without. If there are no arguments, this catch becomes the general catch clause. Flow control is passed to this general catch after the other catches with arguments are exhausted. You can think of the catch statements as similar to a switch (select) statement, with the catch with no arguments as the default (else) choice.

Valid arguments for the catch handlers are either a type or string.

For instance:

```
try {
      a←⍳9
      a[12]
} catch ("INDEX ERROR") {
      ⎕←"index error"
}
```

In this case the catch will evaluate the error thrown to determine if the text string `"INDEX ERROR"` is contained in the error message. If the result is true, then flow control is passed to this handler.

When using types, the default type for all errors is the Exception class.

```
try {
      a←⍳9
      a[12]
} catch ("JUST AN ERROR") {
      ⎕←"didn't happen"
} catch (Exception err) {
      ⎕←err.Message
}
```

In this case flow control passes to the catch with the base Exception type, and the entire error type information is placed in the variable err.

There are many Exception types that can be used for flow control.

The finally choice is always run when included with a try flow control structure. The try is very useful as it can guarantee that the code included in the finally block will always run, regardless of how the function is exited or whether there were errors. For instance, you could include code that closes database connections in the finally block.

```
try {
```

```
      a←⍳9
      return a
} catch {
      ⎕←"an error"
} finally {
      ⎕←"always runs"
}
```

The finally block of code always runs.

23 Throwing exceptions

When execution is proceeding in the try block you can create an exception in two ways:

```
throw ApplicationException("error message")
```

The keyword: throw, will cause the exception class `ApplicationException` to be thrown. Any class which inherits from the Exception class can be used as the exception class.

You can also use:

```
□error "message here"
```

This will cause an exception to be thrown, with the text string which follows error

The Exception class which is used for error is the ApplicationException class.

For instance:

```
try {
      a←1+1
      □error "my error"
} catch ("my error") {
      □←"got an error"
} catch {
      □←"unknown error"
}
```

24 Defining Clean-up Actions

The try statement has the optional clause which is intended to determine what actions are taken when the method is exited. It is executed in all circumstances. For instance:

```
try {
      throw ApplicationException("error")
} finally {
      □←"goodbye"
}
```

A finally clause is executed whether or not an exception has occurred in the try clause. When an exception has occurred, the exception is re-thrown after the finally clause is executed. The finally clause always executes when the try statement is exited, regardless of method.

The code in the finally clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

A try statement must either have one or more except clauses or a finally clause. You can only have one finally clause.

25 Namespace.Class

A namespace is the name which proceeds a class name and is used to categorize a group of classes. A namespace is used only for naming purposes, and has no structure.

For instance:

```
namespace nm1 {
      class cls1 {
      }
      class cls2 {
      }
}
```

Will create two classes named, respectively, `nm1.cls1` and `nm1.cls2.` A namespace is a logical grouping of classes for the purpose of identification and organization.

A class, also called a type, contains both functions (methods) and data. A class can also inherit from another class, making polymorphism possible. All of the methods and data from the inherited class become available on the new class being created.

26 Some definitions

The data in a class can consist of a number of object types, such as a field. From inside the class a field is at the same level as the functions and is global to all functions in the class. It is similar to a global variable from the functions point of view.

Data and methods exist in a class in two primary states. These are instance and static.

27 Instance Objects

A method or field is by default in the instance state. This means that the value of each field exists based on the instance of the class.

An instance of a class is like a private copy of the class. For instance:

```
class cls1 {
      a = 100
      b = 200
}
      myinst1 = cls1()
      myinst2 = cls1()
      myinst1.a = 300
      myinst2.a = 500
      myinst1.a
300
```

The value in myinst1.a is distinct from the value of `myinst2.a` as each of `myinst1` and `myinst2` are distinct instances of the `cls1` class.

Methods which are instance methods use the fields (global variables) in the instance of the class, not the original values of the fields when the class was defined.

For instance:

```
class cls1 {
            a = 100
            b = 200
            function add()  {
                        return a+b
            }
}
            myinst1 = cls1()
            myinst2 = cls1()
            myinst1.a = 300
            myinst2.a = 500
            myinst1.add()
500
            myinst2.add()
700
```

Variables assigned in a function are by default local. To assign a value to a field (global variable) use the "this" keyword.

For instance:

```
function add() {
      this.a = 10
      this.b = 20
      return this.a+this.b
}
```

In the above example the fields a and b have been modified. To create local variables simply create:

```
function add() {
      a = 10
      b = 20
      return a+b+this.a+this.b
}
```

This will add the local variables a and b and the fields a and b. The "this" keyword always refers to the instance of the class.

Referencing a field which has not been overridden by a local variable of the same name does not require the this keyword. However, assignment to the field does require the "this" keyword to differentiate it from a local variable.

For instance:

```
class cls1 {
      a = 100
      b = 200
      function add() {
            a = 10
            c = a+b
            □←a
            □←this.a
            □←c
      }
}
      myinst = cls1()
      myinst.add()
10
100
210
```

The "global" keyword can also be used to identify fields (variables global to the function) within a function:

```
function add() {
      global a
      global b,c,d
      a = 10
      b = 20
}
```

All of the variables, a,b,c and d are fields (global variables) and you do not use the "this" keyword when assigning to them.

It is important to note the following syntax:

```
   a = 100
      function add() {
            a = a
            □←a+100
            a = 200
            □←this.a
            □←a
      }
      add()
200
100
```

```
200
```

The a referenced on the first line references the field named a and assigns it to a local variable named a, a clearer syntax would have been to use:

```
a = this.a
```

28 Static Objects

The static state does not depend on the instance, and is available on the class itself, but not the instance of the class.

For instance:

```
class cls1 {
      public static a = 100
      public static b = 200
      public static function add() {
            return a+b
      }
}
      cls1.a = 10
      cls1.b = 20
      cls1.add()
30
```

The "this" keyword is not available in static functions, as there is no instance. Static fields and methods are therefore available from instance functions, however, instance methods and fields are not available from a static function, as there is no "this" available.

29 Inheritance

A class can inherit from another class. This is the basis of polymorphism.

One of the classes in System.Collections available in the .Net framework is the Hashtable class. This class permits the storage and reference of data by keywords. For instance:

```
a = Hashtable()
a.Add("one", 100)
a.Add("two", 200)
a.Add("three", 300)
```

However, if we want to enhance the Hashtable class to also have an override for the Add method which will only accept a specific range or type of data, we could inherit from Hashtable and then create our Add method.

For instance:

```
class myhash : Hashtable {
      public function Add(key, data) {
            ᗅ first only accept text as the key
            if (82 ≠ □dr key) {
                  □error "Key Error"
            }
            ᗅ accept only integers for data
            if (323 ≠ □dr data) {
                  □error "Data Error"
            }
            Add(key, data)
      }
}

      myinst = myhash()
      myinst.AddInts("test", 10)
      myinst.AddInts("one", 10.3)
```

An exception will be thrown with the message of "Data Error" and □dm will be set to "Data Error"

All of the other methods and fields of the inherited Hashtable will be available on the myhash class.

30 Multiple Inheritance and Access Modifiers

A class can inherit from another class, and from multiple interface classes. An interface class is designed to provide the structure for a base class, but can not be instantiated itself.

Public, Private, Internal

The scope of fields and methods can be scoped to the class, the assembly (group of namespaces in a project) or be made public.

For instance:

```
public class cls1 {
      private a = 100
      internal b = 200
      public c = 300
      public function add() {
            return a+b
      }
      private function minus() {
            return a-b
      }
      internal function times() {
            return a×b
      }
}

      myinst = cls1()
```

The field (global variable) a and the method minus() can only be accessed within the `cls1` class.

The field (global variable) b and the method times() can be accessed within the class or within the same assembly

The field (global variable) c and the method add() can be accessed publicly from other classes and assemblies.

The global keyword when used in a function, creates fields which are by default internal in scope.

The default for all methods and fields is internal.

31 Size of classes

In general, the .Net structure is optimized for the development model where code is broken into numerous smaller classes, instead of one giant class. The best design is to encapsulate a particular functionality in a class or related classes within a namespace and then reuse that functionality.

Creating a single enormous class, with thousands of variables and functions will almost certainly have a disappointing result.

32 Late Binding

Since everything in VisualAPL is an object, then the extension of this is that every thing also has attributes. Most objects will have methods, fields, etc that are available on them, which can be used to manipulate the object or perform some other functionality related to that object.

For instance:

```
   a = ArrayList()
      a.Add(10)
      a[0]
10
```

In this example we created a variable a which is assigned an instance of the ArrayList type. This is a distinct instance, or copy, of the ArrayList type. We then called the Add method on the "a" instance of ArrayList, which added the integer 10 to "a" as the first element. Therefore, a[0] returned that value.

When an instance of an object is created, modification to the instance fields and data properties on the instance effect only that instance. In reality, the code, or methods, of an object are never replicated, only the data objects are replicated.

We could also have done:

```
   a = "test"
      a.IndexOf("st")
2
```

Here, we have created a string object, and then invoked the IndexOf method on that instance of the string object to discover the location of a substring in this string instance.

Since VisualAPL is a dynamic language, any variable can be any object at any moment. Because of this it is possible to create applications very simply and rapidly.

It is also possible to assign the type to a variable, and then use it:

```
   al = ArrayList
      b = al()
      b.Add(10)
      b[0]
10
```

In this case we have assigned the class itself into the variable al, and then created an instance of ArrayList which is assigned to b. You can also create arrays of types:

```
   ahl = ArrayList Hashtable ArrayList HashTable
      b = ahl[0]()
      b.Add(10)
      b[0]
10
```

This provides a very flexible environment to create and manage data, code and solutions.

However, this flexibility has a cost. Since any variable can be anything at any moment, when we perform:

```
a.IndexOf("st")
```

We have to lookup the method IndexOf on whatever type is currently assigned to the variable a. This lookup is not expensive, but at the same time it is not free.

Part of the method lookup can be eliminated by specifying the overload for the method using indexing, for instance:

```
a = "test"
   a.IndexOf[string, int](b, c)
```

In this case, the overload for IndexOf which has as arguments a string and int type is preselected. The values of b and c will be coerced if possible to the respective datatypes, and a runtime error will be thrown if this is not possible. The only lookup required is now based on the type of the variable a. For all types of a, there must be a IndexOf method which takes a string and int as arguments.

When there are many overloads to a method, prespecifying the argument types can improve selection of the correct overloaded method.

If you are going to be doing something a few hundred thousand times a second, late binding is perfect, because of its ease of use and simplicity of creation.

However, if you need the maximum iterations per second, you should look at early binding.

33 Strong Typing a Variable

Strong typing a variable can provide enormous speed improvements in some cases. This means that you can take the 5% of your code which can benefit from typing, and with a few hints, dramatically enhance the performance of your application.

Syntactically this is done as follows:

```
Int32 a = 10
```

The variable a is then under contract to always be a single integer within the scope it was specified. If it was specified in a function, then it is a local in that function and must always be an integer. If it was defined within a class, then it would be an integer field, which means it is a global variable to the functions in the class and must always be a single integer.

Later within the same scope, if you were to try and change the type as follows:

```
string a = "test"
```

This would throw an error during parsing.

A variable can be typed by using the following syntax:

```
String a = "test"
String a
String a, b, c
```

In the last case, all of the variables are set to String type.

For instance, if we create a test function, and create an local variable "a" which is integer, the following happens:

```
   f test() {int a = 10;a = 20;□←a}

      test()

20
```

If we then try to assign a text string to the integer variable a, we see the following error during parsing:

```
   f test() {int a = 10;a = "test";□←a}
```

Invalid Types, can not assign from `System.String` to `System.Int32`

Trying a Double causes the same issue to arise:

```
   f test() {int a = 10;a = 99.5;□←a}
```

Invalid Types, can not assign from `System.Double` to `System.Int32`

However, since this is a dynamic language, you can always create a dynamic variable, and then the data type will be coerced if at all possible during run time. For example, if we assign the Double to the dynamic variable c, and then assign c to a, we see the following:

```
   f test() {int a = 10;c = 99.5;a = c;□←a}
```

```
      test()

100
```

In this case the Double was rounded to an integer and assigned to the strong typed variable a.

If we assign a text string to the intermediate variable c, we then see the following:

```
   ƒ test() {int a = 10;c = "test";a = c;⍞←a}

      test()

116

      ⍞←⎕av⍳'t'

116
```

As you notice, the first element from the text string is coerced to integer and assigned to a. This is similar to doing a ⍞←⎕av⍳'t' as shown.

In all cases, strong types can be assigned to dynamic types, and vice versa. If no conversion exists or is possible, then a type change error is thrown during run time.

In some cases data can be lost in assigning from dynamic types to strong types, as in the instance above where a Double is assigned to an Integer.

In addition you can also specify an array of a given type:

```
Int32[] a = 1 2 3 4
Int32[] b = ⍳10
```

If you want to guarantee strong types on both sides, the following is supported:

```
Int32[] a = new Int32[]{1 2 3 4}
```

You can also initialize an array to the default of the type being created as follows:

```
Int32[] a = new Int32[1000]
```

This will create a strong typed array with 1000 elements each set to 0.

You do not have to use strong typing on the left hand side:

```
a = new Int32[1000]
      ⍴a
1000
```

This provides a very fast mechanism for creating arrays. You can also create matrices using this same syntax:

```
   a = new Double[100,100]
      ⍴a
100 100
```

In this case we have created a matrix which is 100 by 100 and is populated with 0's which are Doubles. Again, this is a very fast way to create an Array, as it uses the direct calls to create the object. It is also valid to use the semicolon in place of the common in the example above, as in:

```
a = new Double[100;100]
```

You can also create arrays of objects, for instance:

```
ArrayList al = new ArrayList[5]
for (int i = 0;i<5;i++) {
      al[i] = new ArrayList()
```

```
}
```
Then all references to al by index will provide a strong typed object, in this case an ArrayList, so:

```
a[0].Add(10)
```
This will use the strong typed ArrayList object and Add the value 10 to the ArrayList.

34 Early Binding

What is early binding, and why is it important?

All objects have the potential to have methods, properties, fields, etc. The way you reference a method on an object is as follows:

```
   myst = "test"
      myst.IndexOf("st")
2
```

The myst object is a string. String objects have the method IndexOf, so we can call the IndexOf method as shown: myst.IndexOf("st"). The result is an index origin 0 index to the first occurrence of the argument string, in this case "st", or a -1 if the argument string does not exist in myst.

With late binding as shown this can be called a several hundred thousand times a second. Which for the vast majority of cases is sufficient.

However, if you absolutely need to access a method the most number of times per second possible, in this example, a couple of million times a second, then you should consider early bound. To make an object early bound, you simply commit that a variable will always be a particular type with a particular shape. For instance, if it is defined as a scalar integer, then it must always be a scalar integer. If it is defined as a integer vector, it must always be an integer vector. Of course, this definition of a variable is limited to its scope, in most cases local to a function.

For our example:

```
   string myst = "test"
      myst.IndexOf("st")
2
```

Now we can call the IndexOf method a couple of million times a second for our string myst. The value of myst can be changed to any string, but it can never be defined as any other type. For instance, myst could never be assigned an integer. Because we can count on myst always being a string in our function, we do not have to wait until the IndexOf function is called, and then lookup what the method IndexOf means for the current type of myst, as myst is guaranteed to be a string type.

When considering early binding, there are really two parts to the equation. The first part is the typing of the variable on which the method will be invoked. The second consideration are the arguments to the method.

In our example, the IndexOf method on an instance of the string type was called with a string `"st"`.

However, we could have called the IndexOf method like this:

```
myst.IndexOf("st",1)
```

This overload method selected of IndexOf would have started evaluation of the string from the index position 1, and would have still returned a 2.

For the IndexOf method on the string type there are these overloads:

String.IndexOf (Char) Reports the index of the first occurrence of the specified Unicode character in this string.

Supported by the .NET Compact Framework.

String.IndexOf (String) Reports the index of the first occurrence of the specified String in this instance.

Supported by the .NET Compact Framework.

String.IndexOf (Char, Int32) Reports the index of the first occurrence of the specified Unicode character in this string. The search starts at a specified character position.

Supported by the .NET Compact Framework.

String.IndexOf (String, Int32) Reports the index of the first occurrence of the specified String in this instance. The search starts at a specified character position.

Supported by the .NET Compact Framework.

String.IndexOf (String, StringComparison) Reports the index of the first occurrence of the specified string in the current String object. A parameter specifies the type of search to use for the specified string.

Supported by the .NET Compact Framework.

String.IndexOf (Char, Int32, Int32) Reports the index of the first occurrence of the specified character in this instance. The search starts at a specified character position and examines a specified number of character positions.

Supported by the .NET Compact Framework.

String.IndexOf (String, Int32, Int32) Reports the index of the first occurrence of the specified String in this instance. The search starts at a specified character position and examines a specified number of character positions.

Supported by the .NET Compact Framework.

String.IndexOf (String, Int32, StringComparison) Reports the index of the first occurrence of the specified string in the current String object. Parameters specify the starting search position in the current string and the type of search to use for the specified string.

Supported by the .NET Compact Framework.

String.IndexOf (String, Int32, Int32, StringComparison) Reports the index of the first occurrence of the specified string in the current String object. Parameters specify the starting search position in the current string, the number of characters in the current string to search, and the type of search to use for the specified string.

Supported by the .NET Compact Framework.

There are over 40 methods in addition to IndexOf on the string object, and most of these methods have many overloads.

As can be seen, when an overloaded method is called, to make early binding possible, the types of the arguments being used to call the method must be clear.

If the argument types are not clear, then the call defaults to late binding. This is quite useful when you want the selection of the method to be decided based on dynamic argument types. For instance, in our example with the IndexOf method the argument might be a string during one call and a char during the next call. Late binding would then choose the method which accepts the string argument one time, and then the method that accepts the character argument the next time.

For example:

```
  string a = "test"
      b = "st"
      a.IndexOf(b)
2
      b = (char) "s"
      a.IndexOf(b)
2
```

By typing the variable a to string, we make early binding possible, however, since our argument can be ambiguous, late binding is selected. In the first case the IndexOf method which accepts a string is called, in the second, the IndexOf which accepts a char is selected.

If we want early binding, we can do the following:

```
sring a = "test"
    string b = "st"
    a.IndexOf(b)
```

In this case we have committed to both the type of "a" and the type of "b", so we can preselect the correct IndexOf method to call.

If you remember, we used the following which created early binding also:

```
string a = "test"
    a.IndexOf("st")
```

The reason this resulted in early binding is because we used a string literal "st" as the argument, so again, a commitment could be made as to which IndexOf method to call, and early binding was possible.

You can also explicitly select the overload by specifying the method arguments as an index.

For instance:

```
a.IndexOf[string, int](b, c)
```

In this case the overload for IndexOf that matches the string and int arguments is selected, and the variables b and c are coerced if possible to string and int respectively. If it is not possible to coerce b and c to the correct data types, then a run time error is thrown.

The idea is that to have early binding you must commit to data types so that the correct method overload is selected. Without the type commitment, then late binding occurs and the correct overload is selected at run time.

You need to remember, that once you have committed to a specific type for a variable, it can not change within its scope. In our example above, an error would be created if we tried to specify "a" as an integer later within our function, once defining it as a string.

Both early and late binding are extremely valuable in development, and provide useful alternatives for invoking methods.

35 Types, why do I care

In most cases, you do not have to worry about datatypes in VisualAPL, as data typing is handled automatically. However, data typing is included in VisualAPL so that you can overtly control and manage the data.

There are many built in data types in .Net. For instance, there are several types of integers, Int32, Int64, UInt32, UInt64, Int16, etc. There are also Double and Single floating point numbers, and Decimal. There are boolean, character, byte, string, and more. There are data collections, like ArrayList, Hashtable, Dictionary, etc. You can even create your own types.

Since the .Net framework is the ultimate in a batteries-included programming environment, understanding types can help you take advantage of all of these tools and utilities. As you use these .Net tools and utilities you will discover a broad range of datatypes, each with its own set of methods, properties, fields, etc.

We will also want to consume methods written in other .Net programming languages. To do this we will also want to understand types.

Since everything in .Net is an object, then our data types are also objects. In fact, when we create a class, it can work as a data object.

It is when we interface to other applications that we will need to concern our selves with types. In particular a method on an object will take specific types and return a specific type.

Again, in most cases, the automatic type handling of VisualAPL will handle the typing, but for occasions when you need to manage the types explicity, VisualAPL includes the tools to both strong type and coerce datatypes.

36 Casting and Coercion of type

Types are extremely specific. An integer scalar type is not an integer array type.

However, VisualAPL has a set of rules for coercing between types. This allows you to cast one type as another if possible.

For instance:

```
  a = 10.3
      b = (int) a
      b
10
```

We have coerced a double to an Int32 integer. We could also cast an integer to a double:

```
   a = (double) b
      a
10
      ⎕dr a
643
      a.GetType()
System.Double
```

It is not possible to create explicitly typed variables in the session. If you want to try this with explicit typing of variables you will need to do that within a function.

All objects have the method `GetType` which returns the current type of the object. This works similar to `⎕dr`, but with `GetType` it is the responsibility of the object to return its type. With `⎕dr` it is the responsibility of a separate function, `⎕dr` in this case, to determine the type of a variable.

It is also possible to cast arrays, for instance:

```
  c = (Double[]) 1 2 3 4 5
      c.GetType()
System.Double[]
```

Since VisualAPL is fundamentally designed to include arrays, you can also simply type:

```
  c = (Double) 1 2 3 4 5
      c.GetType()
System.Double[]
```

While the first instance explicitly states that a vector is to be returned, the second only specifies cast, but does not explicitly indicate that an array will result.

It is also possible to cast a string to integer:

```
a = (int) "abc"
      a
97 98 99
```

However, if you cast the integer array to string, you get the string display of the integers, for instance:

```
a  = (string) 97 98 99
```

```
      a
97 98 99
      a.GetType()
System.String
```

This behavior is compatible with other .Net languages, as casting something to string returns the string representation of the object. All object also have a ToString method which is the method the object uses to display itself. However, the way an object chooses to display itself may be somewhat surprising.

For instance:

```
  a = (int) 1 2 3 4 5
      a.GetType()
System.Int32[]
      a.ToString()
System.Int32[]
```

Even though we might have expected to see the numbers 1 through 5 displayed, the Int32 object chooses to display only its data type. Again, each object chooses how to display itself using the ToString method.

If you had wanted to convert the 97 98 99 back to abc then you could have done two casts:

```
  b = (string) (char) 97 98 99
      b
abc
```

By first casting to characters, we then have the string method on a character array, which then displays the abc result.

We have also included ⎕ucs to convert integers to unicode and unicode to characters:

```
  ⎕ucs "abc"
97 98 99
      ⎕ucs 97 98 99
abc
```

Because all text data in VisualAPL is based on unicode, you can by default display any unicode character, for instance:

```
  ⎕ucs '∈'
8714
```

The APL epsilon is located at character position 8714 in unicode.

When casting to a variable that has been typed, there are several issues to note. First if we type a variable as integer array, this must be done in a function, not in the session, we can use:

```
  Int32[] a = ⍳10
      ⎕←a
0 1 2 3 4 5 6 7 8 9 10
```

If we were to create a as an integer singleton, we could do the following:

```
  Int32 a = ⍳10
      ⎕←a
0
```

In this case "a" can only be an integer singleton, and the first element of the integer array created by ⍳10 is placed in the strong typed variable "a".

Many types will cast to other types, however, it is not possible to cast disparate types to each other, for instance:

```
  a = ⍳10
      b = (Hashtable) a
Invalid cast: System.Collections.Hashtable
```

The error thrown indicates that this is an invalid cast.

Casting is particularly important when calling methods from other assemblies. For instance, with our IndexOf example, we can use castings as follows:

```
  a = (string) 10⍴'hello'
      b = 10.0
      c = 2⍴"el"
      a.IndexOf((string) c, (int) b)
1
```

In this case we took data which was not the types expected by the external IndexOf method on the string type and cast the variable c and b to the correct types for the IndexOf method.

While the automatic type selection process will in most cases choose the correct method, by casting we guarantee that the data is coerced to the types desired for the method overload we want.

37 The Need for Speed

Strong typing can be something of a challenge as it restricts in rather significant ways the manner in which variables and methods can be used, but when you absolutely need to maximize the speed of a function, it is a powerful tool.

When dealing with large matrices or vectors, strong typing will provide only a marginal increase in performance, and in fact in some cases none at all.

However, when dealing with singletons or indexing with scalars it can be quite beneficial. As well as early binding as discussed i the section on Early Binding.

38 Static, Instance and []IO, Random Seed...

Understanding static and instance is critical to using classes. Access Modifiers can only be applied within a class, and will not work in the session.

A method, property, field, etc is defined as static by using the static attribute:

```
public static a = 10
```
This creates a static field named a with a value of 10.

On the next line, you could use:

```
public b = 100
```
This creates an instance field with a value of 100

Both static and instance can exist in the same class, so what is static?

The static portion of a class is really a unique instance of the class, with the fields, properties, etc. initialized in the static constructor. The static constructor of a class is run when the type is first loaded by the CLR, that is the Common Language Runtime, or the system. After that the static portions of a class or type are never reinitialized. This means changes to any static portion of a class results in that change being seen by every program that is referencing that class.

Conversely, the instance field, properties, etc are initialized every time an instance of the class is created.

While this provides a very powerful tool, it is important to understand what this means. If we create a class called myclass:

```
public class myclass {
      public static x = 10
      public y = 100
}
```
Then the first reference to myclass will cause the CLR to run the static constructor, which will set the field x to 10. After that whenever an instance of myclass is created, the field y will be set to 100, but the field x will not be modified.

What this means can best be seen in this example:

```
   f1 = myclass.x
      f1
10
      myclass.x = 200
      myclass.x
200
      a = myclass()
      a.y
100
      myclass.x
200
      a.y = 300
      a.y
300
      b = myclass()
      b.y
100
      myclass.x
200
```

The idea of static and instance becomes very important when considering state values, such as []io and []rl.

The system variables are scoped to the class. This means that setting []io to 1 or 0 in a static method effects all of the static methods, but does not impact instance methods. This is critical, because setting []io in a static method can result in unexpected behaviors if many programs are accessing the static methods of a class. It is simplest to define []io in the static constructor, and then not change it during processing.

This can be done as:

```
⎕io = 0
or
⎕io = 1
```

Then when the static constructor is run the first time, []io is set. The same concept applies to all state variables on static methods, properties, fields, etc. of that class.

In the case of an instance of a class, it is possible to set ⎕io at any time, as it only impacts the particular instance of a class.

Setting ⎕io as a field:

```
⎕io = 0
or
⎕io = 1
```

Will initialize []io in the instance constructors for a class. Then subsequently setting []io during program execution will only effect that instance of the class.

Within a static method it is not possible to set the value of ⎕io for an instance, as no instance exists in the static method. So, setting:

```
static fn1() {
      ⎕io←1
}
```

This will set the static []io to 1, which will impact all static methods, fields, properties, etc being referenced at the time this is set.

It is possible to set the instance value of []io in an instance method also:

```
∱ fn2() {
      ⎕io←1
}
```

Remember that the value of []io for an instance is instance specific, where the []io for statics applies to all references to the static, as there is really only one copy of the static version of the class.