Visual APL Tutorial

This section contains a complete, start to finish tutorial on developing with the Visual APL programming language. It discusses a wide range of language features, as well as intrinsic .NET objects, commonly used objects for daily programming tasks.

1 What is Visual APL?

Visual APLis a next generation APL. Specifically it is APL without lilmits 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.

Visual APLis a .Net language and a peer of C#, VB and managed C++.

Visual APL is integrated with Visual Studio 2005.

Visual APL 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, Visual APL is much more. Visual APL also implements much of the C# language syntax. This means that with Visual APL 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.

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

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

Visual APL 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 Visual APL 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?

Visual APL is APL for .Net. It is not a replication of any particular APL or APL environment. The principle development environment is Visual Studio 2005. This means that development of Visual APL 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, Visual APL 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 Visual APL is trivial. Visual APL uses the unicode standards for all APL characters, supports APL syntax, and provides a copy/paste special to facilitate moving functions. Visual APL can also create objects which are com compatible. This means that Visual APL assemblies can be used in legacy APLs as a com objects. So as you move your APL code to Visual APL, you can use your new .Net dll's from your legacy APL applications.

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

Like legacy APLs, Visual APL 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."

Visual APL 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 Visual APL into compliance with all other .Net languages and makes the integration with those languages transparent when using development systems such as Visual Studio 2005.

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

In Visual APL a variable is local to the function in which it is created unless specified as global. Visual APL 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 Visual APL you will find this object oriented approach both extremely useful and also quite irritating at times.

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

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

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

3 Is Visual APL fast?

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

The examples cover a number of methodologies for both writing code quickly and also creating the fastest solution.

Since the .Net framework represents an advancement by approximately 5 years of all of the legacy APIs, DLL's etc that were found in the Windows type folders, it is safe to assume that most applications will perform better, simply because the underlying calls to databases, gui's, etc are dramatically faster. In addition, accessing these resources is now native to Visual APL, and does not require a translation interface.

Visual APL also supports adding typing to your functions should you have a function which needs to perform as quickly as possible. This type of coding usually occurs in less than 5% of an application, but with Visual APL, typing is native and not an exception.

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 Visual APL, 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 been proprietary and would only work with a particular APL interpreter. As a .Net language Visual APL source is kept in unicode text files which any editor can open and modify.

This is one of the principals of .Net and Visual Studio. Everything is open and readable, nothing is proprietary.

6 What does the source file look like?

The basic construct 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. You might also have:

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

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

The reason for this is that each dll or exe is only granted access to those resources it specifically requests when being created as part of the managed and verifiable framework.

7 Is there an Interpreter?

Yes, Visual APL still provides code interpretation on-the-fly.

You can still ⎕def a function while you are running code.

You can run ⍎ 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 Visual APL Tutorial

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

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

In fact you should discover that you can use almost all of your APL code as well as the vast majority of C# examples. 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 Visual APL executes 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 Visual APL programming language.

10 Using Visual APL as a desktop calculator

To display your session in Visual Studio, select the View Menu

Then Other Windows

Then Cielo Explorer

The session should remain open between closing and opening Visual Studio. You only need to reopen the session should you close it by pressing the x close box in the upper right hand corner of the session, or between reinstalls of Visual APL.

The session is independent from any project or solution you are currently editing or viewing in Visual Studio.

The session should appear very much like any APL session you have encountered.

For instance:

```
      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 broken up into parts, 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 Visual APL 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 ← 0 in

your session will not effect the way the methods which belong to other objects work. But it will effect the APL operators in your Assembly.

11 APL Operators and Functions

In general the APL operators and functions included in Visual APL 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 done an approximate equals, based on □ct. The APL equal symbol has always been accessed using the right alt/5 key combination. In the operator set used by Visual APL the unicode APL symbol for approximately equal (≈) is used for APL equality comparison.

The ascii = symbol is used for assign by reference, as it is in all other computer languages.

The ← assign symbol is assign by value, the same as always in APL.

Having the new = assign by reference provides some powerful options. Remembering that everything is an object in Visual APL, 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?      "     "
```

Implied line continuation occurs 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
5
      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.MaxValule
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 these you would include the following at the top of your file, run it in the session:

13 Using System.Collections

The .Net framework includes many new data types. The Collections namespce 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, you must add:

using System.Collections

To either your project or execute the line in your 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:

```
      fm ⎕wi 'Create' 'Form' '' '' '' ''

fm

      fm.b ⎕wi 'Create' 'Button' '' '' '' ''

fm.b

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

       push(a,b) {⎕←a;⎕←b}      ƒ

      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' '' ''

        push(a,b) {⎕←a;⎕←b}      ƒ

      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
```

If you noticed above we created a function using a function signature which might be new to some APL programmers. You can review the new method creations in that section.

14 Controlling Program Flow

Normally statements in Visual APL are executed one after another in the order they were written. This sequential order is the default. Control Flow statements alter the flow of a program.

Using the conditional if statement

There are two structures for the if statement, one is the typical APL syntax and the other is the C# structure:

```
: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 of course optional.

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

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

This 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                    "    "
}
```

Will display:

```
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 Visual APL 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, Visual APL 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
}
```

Visual APL also includes a new function definition character which is created using right alt/f and displays as the mathematical symbol for function : f

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

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

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

```
myfn =   fn3(a, b, c) {              f
      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   outerfn(a) {
      return   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   (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  9 9.4)  "              "
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.cls1.` A namespace is a logical grouping of classes for the purpose of naming.

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) {
            a first only accept text as the key
            if (82 ≠ dr key) {
                  error  Key Error              "          "
            }
            a 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 Visual APL 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 Visual APL 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:

```
        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:

```
        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:

```
        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:

```
        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:

```
          test() {int a = 10;c =  t∮st ;a = c;□←a}       "     "

      test()

116

      □a⍣l'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 □a⍣l'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 Visual APL, as data typing is handled automatically. However, data typing is included in Visual APL 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 Visual APL will handle the typing, but for occasions when you need to manage the types explicity, Visual APL 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, Visual APL 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 Visual APL 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 Visual APL 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 ⍳⎕IO 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() {                    f
      □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.

Visual APL Development Environment (Integrated with Visual Studio)

Introduction Visual APL

The Visual APL integrated with Visual Studio is a collection of development tools exposed through a Visual Studio interface. Some of the tools are shared with other Visual Studio languages, and some, such as the Visual APL compiler, are unique to Visual APL. This documentation provides an overview of how to use the most important Visual APL tools as you work in Visual Studio in various phases of the development process.

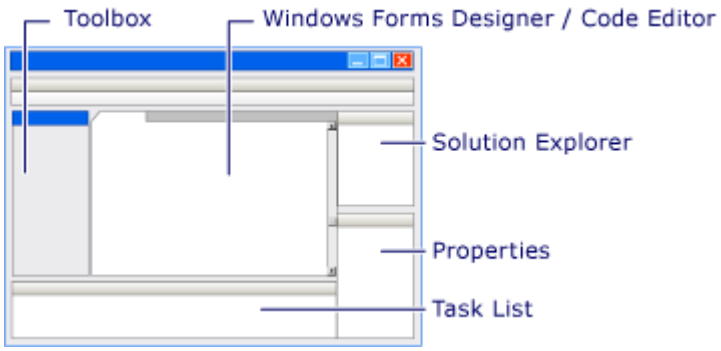
Visual APL Tools

The following are the most important tools and windows in Visual APL. The windows for most of these tools can be opened from the **View** menu.

- The Code Editor, for writing source code.
- The Visual APL compiler, for converting Visual APL source code into an executable program.
- The Visual Studio debugger, for testing your program.
- The **Toolbox** and **Designer**, for rapid development of user interfaces using the mouse.
- **Solution Explorer**, for viewing and managing project files and settings.
- **Project Designer**, for configuring compiler options, deployment paths, resources, and more.
- **Class View**, for navigating through source code according to types, not files.
- **Properties Window**, for configuring properties and events on controls in your user interface.
- Object Browser, for viewing the methods and classes available in dynamic link libraries including .NET Framework assemblies and COM objects.
- Document Explorer, for browsing and searching product documentation on your local machine and on the Internet.

Visual Studio Exposes Tools Overview

You can interact with the VS tools through windows, menus, property pages, and wizards. The basic VS looks something like this:



You can quickly access any open tool windows or files by pressing CTRL + TAB. For more information, see Navigating and Searching (Visual APL).

Editor and Windows Form Designer Windows

The large main window is used by both the Code Editor and the Windows Forms Designer. You can toggle between code view and Design view by pressing F7, or clicking **Code** or **Designer** on the **View** menu. While in Design view, you can drag controls onto the window from the **Toolbox**, which you can make visible by clicking on the **Toolbox** tab on the left margin. For more information about the Code Editor, see Editing Code (Visual APL). For more information about the Windows Forms Designer, see Windows Forms Designer.

The **Properties** window in the lower right is populated only in Design view. It enables you to set properties and hook up events for user interface controls such as buttons, text boxes, and so on. When you set this window to **Auto Hide**, it will collapse into the right margin whenever you switch to **Code View**. For more information about the **Properties** window and the Designer, see Designing a User Interface (Visual APL).

Solution Explorer and Project Designer

The window in the top right is **Solution Explorer**, which shows all the files in your project in a hierarchical tree view. When you use the **Project** menu to add new files to your project, you will see them reflected in **Solution Explorer**. In addition to files, **Solution Explorer** also displays your project settings, and references to external libraries required by your application.

The **Project Designer** property pages are accessed by right-clicking on the **Properties** node in **Solution Explorer**, and then clicking **Open**. Use these pages to modify build options, security requirements, deployment details, and many other project properties. For more information about **Solution Explorer** and the **Project Designer**, see Creating a Project (Visual APL).

Compiler, Debugger, and Error List Windows

The Visual APL compiler has no window because it is not an interactive tool, but you can set compiler options in the **Project Designer**. When you click **Build** on the **Build** menu, the Visual APL compiler is invoked by the IDE. If the build is successful, the status pane displays a Build Succeeded message. If there were build errors, the **Error List** window appears below the editor/designer window with a list of errors. Double-click an error to go to the problem line in your source code. Press F1 to see Help documentation for the highlighted error.

The debugger has various windows that display values of variables and type information as your application is running. You can use the Code Editor window while debugging to specify a line at which to pause execution in the debugger, and to step through code one line at a time. For more information, see Building and Debugging (Visual APL).

Customizing the IDE

All of the windows in Visual APL can be made dockable or floating, hidden or visible, or can be moved to new locations. To change the behavior of a window, click the down arrow or push-pin icons on the title bar and select from among the available options. To move a docked window to a new docked location, drag the title bar until the window dropper icons appear. While holding down the left mouse button, move the mouse pointer over the icon at the new location. Position the pointer over the left, right, top or bottom icons to dock the window on the specified side. Position the pointer over the middle icon to make the window a tabbed window. As you position the pointer, a blue semi-transparent rectangle appears, which indicates where the window will be docked in the new location.

-- Operator (Visual APL Reference)

The decrement operator (--) decrements its operand by 1. The decrement operator can appear only after its operand:

Remarks

The postfix decrement operation. The result of the operation is the value of the operand before it has been decremented.

Numeric and enumeration types have predefined increment operators. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn()
{
    x = 1.5;
    print x--;
    print x
    x = 1.5 10 20;
    print x--;
    print x;
}
```

Output

```
1.5
0.5
1.5 10 20
0.5 9 19
```

-- Operator (Visual APL Reference)

The decrement operator (--) decrements its operand by 1. The decrement operator can appear only after its operand:

Remarks

The postfix decrement operation. The result of the operation is the value of the operand before it has been decremented.

Numeric and enumeration types have predefined increment operators. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn()
{
    x = 1.5;
    print x--;
    print x
    x = 1.5 10 20;
    print x--;
    print x;
}
```

Output

```
1.5
0.5
1.5 10 20
0.5 9 19
```

>> Operator (Visual APL Reference)

The right-shift operator (>>) shifts its first operand right by the number of bits specified by its second operand.

Remarks

If the first operand is an int or uint (32-bit quantity), the shift count is given by the low-order five bits of the second operand (second operand & 0x1f).

If the first operand is a long or ulong (64-bit quantity), the shift count is given by the low-order six bits of the second operand (second operand & 0x3f).

If the first operand is an int or long, the right-shift is an arithmetic shift (high-order empty bits are set to the sign bit). If the first operand is of type uint or ulong, the right-shift is a logical shift (high-order bits are zero-filled).

User-defined types can overload the >> operator; the type of the first operand must be the user-defined type, and the type of the second operand must be int. For more information, see operator. When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded.

Example

```
using System;
function fn()
{
    i = -1000;
    print i >> 3;
}
```

Output

```
-125
```

>= Operator (Visual APL Reference)

All numeric and enumeration types define a "greater than or equal" relational operator, >= that returns true if the first operand is greater than or equal to the second, false otherwise.

Remarks

User-defined types can overload the >= operator. For more information, see operator. If >= is overloaded, <= must also be overloaded. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn() {
    print 1.1 >= 1;
    print 1.1 >= 1.1;
}
```

Output

```
true
true
```

== Operator (Visual APL Reference)

For predefined value types, the equality operator (==) returns true if the values of its operands are equal, false otherwise. For reference types other than string, == returns true if its two operands refer to the same object. For the string type, == compares the values of the strings.

Remarks

User-defined value types can overload the == operator (see operator). So can user-defined reference types, although by default == behaves as described above for both predefined and user-defined reference types. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn() {
    // Numeric equality:  True
    print (2 + 2) == 4;

    // Reference equality:  different objects,
    // same boxed value:  true.
    s = 1;
    t = 1;
    print s == t;

    // Define some strings:
    a =  hello ;                "     "
    b =  String.Copy(a);
    c =  hello ;                "     "

    // Compare string values of a constant and an instance:  True
    print a == b;
    print a == c

}
```

Output

```
true
true
true
true
```

<= Operator (Visual APL Reference)

All numeric and enumeration types define a "less than or equal" relational operator (<=) that returns true if the first operand is less than or equal to the second, false otherwise.

Remarks

User-defined types can overload the <= operator. For more information, see operator. If <= is overloaded, >= must also be overloaded. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn() {
    print 1 <= 1.1
    print 1.1 <= 1.1
}
```

Output

```
true
true
```

<< Operator (Visual APL Reference)

The left-shift operator (<<) shifts its first operand left by the number of bits specified by its second operand. The type of the second operand must be an int.

Remarks

If first operand is an int or uint (32-bit quantity), the shift count is given by the low-order five bits of second operand.

If first operand is a long or ulong (64-bit quantity), the shift count is given by the low-order six bits of second operand.

The high-order bits of first operand are discarded and the low-order empty bits are zero-filled. Shift operations never cause overflows.

User-defined types can overload the << operator (see operator); the type of the first operand must be the user-defined type, and the type of the second operand must be int. When a binary operator is overloaded, the corresponding assignment operator, if any, is also implicitly overloaded.

Example

```
using System;
function fn()
{
    i = 1;
    lg = 1L;
    print string.Format( 0x{0:x} , i << 1);     "       "
    Console.WriteLine( 0x{0:x} , i << 33);    "       "
    Console.WriteLine( 0x{0:x} , lg << 33);   "       "
}
```

Output

```
0x2
0x2
0x200000000
Comments
```

Note
The i<<1 and i<<33 give the same result, because 1 and 33 have the same low-order five bits.

++ Operator (Visual APL Reference)

The increment operator (++) increments its operand by 1. The increment operator can appear only after its operand:

Remarks

The postfix increment operation. The result of the operation is the value of the operand before it has been incremented.

Numeric and enumeration types have predefined increment operators. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn()
{
    x = 1.5;
    print x++;
    print x
    x = 1.5 10 20;
    print x++;
    print x;
}
```

Output

```
1.5
2.5
1.5 10 20
2.5 11 21
```

|| Operator (Visual APL Reference)

The conditional-OR operator (||) performs a logical-OR of its bool operands, but only evaluates its second operand if necessary.

Remarks

The operation

x || y

corresponds to the operation

x | y

except that if x is true, y is not evaluated (because the result of the OR operation is true no matter what the value of y might be). This is known as "short-circuit" evaluation.

The conditional-OR operator cannot be overloaded, but overloads of the regular logical operators and operators true and false are, with certain restrictions, also considered overloads of the conditional logical operators.

Example

In the following example, observe that the expression using || evaluates only the first operand.

```
using System;
function Method1()
{
    print  Method1 called ;       "               "
    return true;
}

static bool Method2()
{
    print  Method2 called ;       "               "
    return false;
}

static void Main()
{
    print  regular OR: ;          "            "
    print string.Format( result is {0} , Method1"() ¦ Method2()");
    print  short-circuit OR: ;    "                   "
    print string.Format( result is {0} , Method1"() ¦¦ Method2("));
}
```

Output

```
regular OR:
Method1 called
Method2 called
result is true
short-circuit OR:
Method1 called
result is true
```

| Operator (Visual APL Reference)

Binary | operators are predefined for the integral types and bool. For integral types, | computes the bitwise OR of its operands. For bool operands, | computes the logical OR of its operands; that is, the result is false if and only if both its operands are false.

Remarks

The | operator evaluates both operators regardless of the first one's value. For example:

Example

```
using System;
function fn() {
    print true ¦ false;  // logical or
    print false ¦ false; // logical or
    print string.Format( 0x{0:x} , 0xf8 ¦ 0x3f);"   // bi"twise or
}
```

Output

```
True
False
0xff
```

^ Operator (Visual APL Reference)

Binary ^ operators are predefined for the integral types and bool. For integral types, ^ computes the bitwise exclusive-OR of its operands. For bool operands, ^ computes the logical exclusive-or of its operands; that is, the result is true if and only if exactly one of its operands is true.

Remarks

Operations on integral types are generally allowed on enumeration.

Note
This operator is created with the shift-6 key, not to be confused with the alt-0 key which creates the array and operator.

Example

```
using System;
function fn()
{
    print true ^ false;  // logical exclusive-or
    print false ^ false; // logical exclusive-or
    // Bitwise exclusive-or:
    print string.Format( 0x{0:x} , 0xf8 ^ 0x3f);"       "
}
```

Output

```
True
False
0xc7
```

&& Operator (Visual APL Reference)

The conditional-AND operator (&&) performs a logical-AND of its bool operands, but only evaluates its second operand if necessary.

Remarks

The operation

x && y

corresponds to the operation

x & y

except that if x is false, y is not evaluated (because the result of the AND operation is false no matter what the value of y may be). This is known as "short-circuit" evaluation.

The conditional-AND operator cannot be overloaded, but overloads of the regular logical operators and operators true and false are, with certain restrictions, also considered overloads of the conditional logical operators.

Example

In the following example, observe that the expression using && evaluates only the first operand.

```
using System;
function Method1()
{
   print  Method1 called ;       "               "
   return false;
}

function Method2()
{
   print  Method2 called ;       "               "
   return true;
}

function fn()
{
   print  regular AND: ;         "             "
   print string.Format( result is {0} , Method1"() & Method2()");
   print  short-circuit AND: ;   "                    "
   print string.Format ( result is {0} , Method1"() && Method2(""));
}
```

Output

```
regular AND:
Method1 called
Method2 called
result is false
short-circuit AND:
Method1 called
result is false
```

& Operator (Visual APL Reference)

The & operator is a binary operator and works only on scalars.

Remarks

Binary & operators are predefined for the integral types and bool. For integral types, & computes the logical bitwise AND of its operands. For bool operands, & computes the logical AND of its operands; that is, the result is true if and only if both its operands are true.

The & operator evaluates both operators regardless of the first one's value.

For example:

```
int i = 1;
if (false & i == 1)
{
    // i is incremented, but the conditional
    // expression evaluates to false, so
    // this block does not execute.
}
```

Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn() {
    print  true & false; // logical and
    print  true & true ;  // logical and
    print string.Format(0x{0:x} , 0xf8 & 0x3f);  // bitwise and
}
```

Output

```
False
True
0x38
```

!= Operator (Visual APL Reference)

The inequality operator (!=) returns false if its operands are equal, true otherwise. Inequality operators are predefined for all types, including string and object. User-defined types can overload the != operator.

Remarks

For predefined value types, the inequality operator (!=) returns true if the values of its operands are different, false otherwise. For reference types other than string, != returns true if its two operands refer to different objects. For the string type, != compares the values of the strings.

User-defined value types can overload the != operator (see operator). So can user-defined reference types, although by default != behaves as described above for both predefined and user-defined reference types. Operations on integral types are generally allowed on enumeration.

Example

```
using System;
function fn() {
    // Numeric inequality:
    print (2 + 2)  = 4;           !

    // Reference equality:  two objects, same boxed value
    s = 1;
    t = 1;
    print s  = t);          !

    // String equality:  same string value, same string objects
    a =  hello ;               "     "
    b =  hello ;               "     "

    // compare string values
    print a  = b;           !

}
```

Output

```
false
false
false
```

Visual APL Keywords

Keywords are predefined reserved identifiers that have special meanings to the compiler. They cannot be used as identifiers in your program.

abstract	as	and	base
bit	bool	break	byte
case	catch	char	class
continue	decimal	default	definition
delegate	do	double	else
elseif	enum	event	false
finally	float	for	foreach
function	get	global	goto
if	in	int	intn
interface	is	long	namespace
new	nop	null	not
object	operator	or	out
override	params	print	private
property	protected	public	readonly
ref	refbyfile	refbyname	repeat
return	sbyte	sealed	set
short	sizeof	static	step
string	switch	then	this
throw	to	true	try
typeof	uint	ulong	until
ushort	using	value	virtual
void	volatile	while	yield

Exception Handling Statements (Visual APL Reference)

Visual APL provides built-in support for handling anomalous situations, known as exceptions, which may occur during the execution of your program. These exceptions are handled by code that is outside the normal flow of control.

The following exception handling topics are explained in this section:

```
throw
try-catch
try-finally
try-catch-finally
```

throw (Visual APL Reference)

The throw statement is used to signal the occurrence of an anomalous situation (exception) during the program execution.

Remarks

The thrown exception is an object whose class is derived from System.Exception, for example:

```
class MyException :  System.Exception {}
// ...
throw new MyException();
```

Usually the throw statement is used with try-catch or try-finally statements. When an exception is thrown, the program looks for the catch statement that handles this exception.

You can also rethrow a caught exception using the throw statement. For more information and examples, see try-catch and Throwing Exceptions.

Example

This example demonstrates how to throw an exception using the throw statement.

```
using System;
function fn() {
    s = null;

    if (s == null) {
        throw ArgumentNullException( Error thrown );        "             "
    }

    print  The string s is null ; "// not executed      "
}

Output

Error thrown
```

try-catch (Visual APL Reference)

The try-catch statement consists of a try block followed by one or more catch clauses, which specify handlers for different exceptions.

Remarks

The try block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully. For example, the following attempt to cast throws an error.

```
function fn() {
    int a = 10
    try {
        a = Form()
    } catch (Exception e) {
        print  this is the error     "                "
        print e.Message
    }
}
```

Output

```
this is the error
Unable to cast object of type 'System.Windows.Forms.Form' to type
'System.IConvertible'.
```

The catch clause can be used without arguments, in which case it catches any type of exception, and referred to as the general catch clause. It can also take an object argument derived from System.Exception, in which case it handles a specific exception. For example:

```
catch (InvalidCastException e)
{
}
```

It is possible to use more than one specific catch clause in the same try-catch statement. In this case, the order of the catch clauses is important because the catch clauses are examined in order. Catch the more specific exceptions before the less specific ones.

A throw statement can be used in the catch block to re-throw the exception, which has been caught by the catch statement. For example:

```
catch (InvalidCastException e)
{
    throw (e);     // Rethrowing exception e
}
```

If you want to re-throw the exception currently handled by a parameter-less catch clause, use the throw statement without arguments. For example:

```
catch
{
    throw;
}
```

Example

In this example, the try block contains a call to the method MyMethod() that may cause an exception. The catch clause contains the exception handler that simply displays a message on the screen. When the throw statement is called from inside MyMethod, the system looks for the catch statement and displays the message Exception caught.

```
using System;
function ProcessString(s)
    {
    if (s == null)
        {
        throw new ArgumentNullException( Argument is null );    "
    }
}

function fn()
    {
    try
        {
        s = null;
        ProcessString(s);
    }
    catch (Exception e)
        {
        print string.Format( {0} Exception caught. ," e.Message);          "
    }
}
```

Output

```
      fn()
Argument is null Exception caught.
```

try-finally (Visual APL Reference)

The finally block is useful for cleaning up any resources allocated in the try block as well as running any code that must execute even if there is an exception. Control is always passed to the finally block regardless of how the try block exits.

Remarks

Whereas catch is used to handle exceptions that occur in a statement block, finally is used to guarantee a statement block of code executes regardless of how the preceding try block is exited.

Example

In this example, there is one invalid conversion statement that causes an exception. When you run the program, you get a run-time error message, but the finally clause will still be executed and display the output.

```
// try-finally
using System;
function fn() {
    i = 1 2 3;
    try {
        // Invalid index
        a = i[5];
    } finally {
        print string.Format( i = {0} , i);          "       "
    }
}

      fn()
i =  System.Int32[]
Index was outside the bounds of the array.
```

Comments

Although an exception was caught, the output statement included in the finally block will still be executed.

Jump Statements (Visual APL Reference)

Branching is performed using jump statements, which cause an immediate transfer of the program control. The following keywords are used in jump statements:

```
break
continue
goto
return
throw
```

Branch (→) (Visual APL Reference)

The Branch (→) statement transfers the program control to a labeled statement, either by a dynamic selection of the label to branch to, or directly to the specified label.

Syntax

```
    → label
```

label: A label value.

Remarks

In most .Net languages, a goto statement is the only method provided for altering program flow based on labels in a function, and usually this statement requires that its right argument is a single literally specified label. Visual APL fully supports this construct via the goto statement.

The Branch statement is the dynamic complement to the static goto statement, in that the argument to the Branch statement can be any expression which returns a valid label within the function in which the Branch statement has been called.

Label Values

Labels in all .Net languages are in actuality just integers (or System.Int32).

The label value which is passed to the Branch statement is therefore a simple Integer, which is the index of the label in the function to which code flow will be transferred.

There are three categories of label values which can be passed to the Branch statement:

1. A value of 0. If a 0 is passed to the branch statement, the function returns immediately, returning either the value of the default return variable of the function, or if there is no default return variable declared, the value **null** is returned.
2. A value which is one of the declared labels in the function.
   If labels L1 and L2 are sequentially declared in a function, then L1 will be assigned the value 1, and L2 will be assigned the value 2. In this function, the Branch statement will transfer control to L1 if its argument is the number 1, and L2 if the argument is the number 2.
3. Values greater then the number of declared labels. Any integer argument to the Branch statement which is greater then the number of declared label statements in the function will have no branching effect, meaning that code flow will be transferred to the next statement after the Branch statement in the function, and has the net result as a **nop** in the function.

Example

```
function fn(a) {
    if (a == 10) {
        lb = L2
    } else {
        lb = L1
    }

    // branch to the specified statement
    →lb
```

```
    print  dont show this text    "                    "

    L1:
    print  a was not 10           "             "
    L2:
    print  end                    "   "
}

      fn(10)
end
      fn(11)
a was not 10
end
```

break (Visual APL Reference)

The break statement terminates the closest enclosing loop or switch statement in which it appears. Control is passed to the statement that follows the terminated statement, if any.

Example

In this example, the conditional statement contains a counter that is supposed to count from 1 to 100; however, the break statement terminates the loop after 4 counts.

```
using System;
function fn() {
    for (i = 1; i <= 100; i++) {
        if (i == 5)
        {
            break;
        }
        print i;
    }
}

 Output
1
2
3
4
This example demonstrates the use of break in a switch statement.

using System;
function fn(n) {
    switch (n)
    {
        case 1:
            print string.Format( Current value is {0} , "1);                  "
            break;
        case 2:
            print string.Format( Current value is {0} , "2);                  "
            break;
        case 3:
            print string.Format( Current value is {0} , "3);                  "
            break;
        default:
            print string.Format( default selection. );  "                   "
            break;
    }
}

      fn(1)
Current value is 1
      fn(2)
Current value is 2
      fn(3)
Current value is 3
      fn(4)
default selection.
```

continue (Visual APL Reference)

The continue statement passes control to the next iteration of the enclosing iteration statement in which it appears.

Example

In this example, a counter is initialized to count from 1 to 10. By using the continue statement in conjunction with the expression (i < 9), the statements between continue and the end of the for body are skipped.

```
using System;
function fn() {
    for (i = 1; i <= 10; i++) {
        if (i < 9) {
            continue;
        }
        print i;
    }
}

Output
9
10
```

goto (Visual APL Reference)

The goto statement transfers the program control directly to a labeled statement.

Remarks

A common use of *goto* is to transfer control to a specific switch-case label or the default label in a switch statement.

The *goto* statement is also useful to get out of deeply nested loops.

For dynamic and conditional branching, see the Branch statement.

Example

```
function fn(a) {
    if (a == 10) {
        goto L1
    }
    print  a was not 10          "             "
    L1:
    print  end                   "   "
}

      fn(10)
end
      fn(11)
a was not 10
end
```

return (Visual APL Reference)

The return statement terminates execution of the method in which it appears and returns control to the calling method. It can also return an optional value. If the method is a void type, the return statement can be omitted.

Example

In the following example, the method CalculateArea() returns the variable Area as a double value.

```
using System;
function CalculateArea(r) {
    area = r × r × Math.PI;
    return area;
}

function fn() {
    radius = 5;
    r = CalculateArea(radius)
    a = string.Format( The area is {0:0.00} ,"r);                    "
    print a
}

     fn()
The area is 78.54
```

Selection Statements (Visual APL Reference)

A selection statement causes the program control to be transferred to a specific flow based upon whether a certain condition is true or not.

The following keywords are used in selection statements:

```
if
else
elseif
switch
case
default
```

foreach, in (Visual APL Reference)

The foreach statement repeats a group of embedded statements for each element in an array or an object collection. The foreach statement is used to iterate through the collection to get the desired information, but should not be used to change the contents of the collection to avoid unpredictable side effects.

Remarks

The embedded statements continue to execute for each element in the array or collection. After the iteration has been completed for all the elements in the collection, control is transferred to the next statement following the foreach block.

At any point within the foreach block, you can break out of the loop using the break keyword, or step directly to the next iteration in the loop by using the continue keyword.

A foreach loop can also be exited by the goto, return, or throw statements.

Example

In this example, foreach is used to display the contents of an array of integers.

```
function fn(a) {
    foreach (i in a) {
        print i
    }
}

      fn(⍳3)
0
1
2
      fn( one   two )            "   " "   "
one
two
```

The foreach also supports multiple variables:

```
function fn() {
    foreach (a b c in (1 2 3) (4 5 6)) {
        print a
        print b
        print c
    }
}

      fn()
1
2
3
4
5
6
```

if-else (Visual APL Reference)

The if statement selects a statement for execution based on the value of a Boolean expression. In the following example a Boolean flag flagCheck is set to true and then checked in the if statement. The output is: The flag is set to true.

Example

```
function fn(flagCheck) {
    if (flagCheck == true) {
        print  "The flag is set to true". ;                    "
    } else {
        print  "The flag is set to false". ;                     "
    }
}
```

Remarks

If the expression in the parenthesis is evaluated to be true, then the print "The boolean flag is set to true."; statement is executed. After executing the if statement, control is transferred to the next statement. The else is not executed in this example.

If you wish to execute more than one statement, multiple statements can be conditionally executed by including them into blocks using {} as in the example above.

The statement(s) to be executed upon testing the condition can be of any kind, including another if statement nested into the original if statement. In nested if statements, the else clause belongs to the last if that does not have a corresponding else.

For example:

```
function fn(x,y) {
    if (x > 10) {
        if (y > 20) {
            print  Statement_1 ;          "           "
        } else {
            print  Statement_2 ;          "           "
        }
    } else if (x < 5) {
        print  Statement_3
    }
}
```

In this case, Statement_2 will be displayed if the condition (x > 10) evaluates to false

If x is less than 5 then Statement_3 will be displayed.

switch-case (Visual APL Reference)

The switch statement provides a method for controlling the code flow based on the Identity comparison of a value to a series of values.

```
function fn(a) {
    switch (a) {
        case 10:
        print  10                    "   "
        break
        case  test :                "     "
        print  test                  "     "
        break
        default:
        print  default               "        "
        break
    }
}

      fn(10)
10
      fn(  test  )               "     "
test
      fn(100)
default
```

Iteration Statements (Visual APL Reference)

You can create loops by using the iteration statements. Iteration statements cause embedded statements to be executed a number of times, subject to the loop-termination criteria. These statements are executed in order, except when a jump statement is encountered.

The following keywords are used in iteration statements:

```
do
for
foreach
in
while
```

do (Visual APL Reference)

The do statement executes a statement or a block of statements enclosed in {} repeatedly until a specified expression evaluates to false. In the following example the do-while loop statements execute as long as the variable y is less than 5.

Example

```
using System;
function fn() {
    x = 0;
    do {
        print x;
        x++;
    } while (x < 5);
}
```

Output

```
0
1
2
3
4
```

Remarks

Unlike the while statement, a do-while loop is executed once before the conditional expression is evaluated.

At any point within the do-while block, you can break out of the loop using the break statement. You can step directly to the while expression evaluation statement by using the continue statement; if the expression evaluates to true, execution continues at the first statement in the loop. If the expression evaluates to false, execution continues at the first statement after the do-while loop.

A do-while loop can also be exited by the goto, return, or throw statements.

for (Visual APL Reference)

The for loop executes a statement or a block of statements repeatedly until a specified expression evaluates to false. The for loop is handy for iterating over arrays and for sequential processing. In the following example, the value of int i is written to the session and i is incremented each time through the loop by 1.

Example

```
using System;
function fn()
{
    for (i = 1; i <= 5; i++)
    {
        print i
    }
}
```

Output

```
1
2
3
4
5
```

Remarks

The for statement executes the enclosed statement or statements repeatedly as follows:

First, the initial value of the variable i is evaluated.

Then, while the value of i is less than or equal to 5, the condition evaluates to true, the print statement is executed and i is reevaluated.

When i is greater than 5, the condition becomes false and control is transferred outside the loop.

Because the test of conditional expression takes place before the execution of the loop, therefore, a for statement executes zero or more times.

All of the expressions of the for statement are optional; for example, the following statement is used to write an infinite loop:

```
for (;;)
{
    // ...
}
```

The for statement also supports an else block. The else block is only evaluated if the for block is never evaluated.

Example:

```
function fn(a) {
    for (i=0;i<a;i++) {
        print i
    } else {
        print  no iteration          "            "
    }
```

```
}

      fn(3)
0
1
2
      fn(0)
no iteration
```

while (Visual APL Reference)

The while statement executes a statement or a block of statements until a specified expression evaluates to false.

Example

```
using System;
function fn(a)  {
    n = 1;
    while (n < a)
    {
        print string.Format( Current value of n is {"0} , n);                "
        n++;
    }
}

      fn(3)
Current value of n is 1
Current value of n is 2
```

The while statement also supports an else block. In the event the while loop is never entered then the else block will be evaluated.

Example:

```
function fn(a) {
    n = 1;
    while (n < a)
        {
        print string.Format( Current value of n is {"0} , n);                "
        n++;
    } else {
        print  never entered         "             "
    }
}

      fn(3)
Current value of n is 1
Current value of n is 2
      fn(1)
never entered
```

A while loop can be terminated when a break, goto, return, or throwstatement transfers control outside the loop. To pass control to the next iteration without exiting the loop, use the continuestatement. Notice the difference in output in the three previous examples, depending on where int n is incremented. In the example below no output is generated.

Exception Handling Statements (Visual APL Reference)

Visual APL provides built-in support for handling anomalous situations, known as exceptions, which may occur during the execution of your program. These exceptions are handled by code that is outside the normal flow of control.

The following exception handling topics are explained in this section:

```
throw
try-catch
try-finally
try-catch-finally
```

throw (Visual APL Reference)

The throw statement is used to signal the occurrence of an anomalous situation (exception) during the program execution.

Remarks

The thrown exception is an object whose class is derived from System.Exception, for example:

```
class MyException :  System.Exception {}
// ...
throw new MyException();
```

Usually the throw statement is used with try-catch or try-finally statements. When an exception is thrown, the program looks for the catch statement that handles this exception.

You can also rethrow a caught exception using the throw statement. For more information and examples, see try-catch and Throwing Exceptions.

Example

This example demonstrates how to throw an exception using the throw statement.

```
using System;
function fn() {
    s = null;

    if (s == null) {
        throw ArgumentNullException( Error thrown );        "             "
    }

    print  The string s is null ; "// not executed      "
}

Output

Error thrown
```

try-catch (Visual APL Reference)

The try-catch statement consists of a try block followed by one or more catch clauses, which specify handlers for different exceptions.

Remarks

The try block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully. For example, the following attempt to cast throws an error.

```
function fn() {
    int a = 10
    try {
        a = Form()
    } catch (Exception e) {
        print  this is the error     "                "
        print e.Message
    }
}
```

Output

```
this is the error
Unable to cast object of type 'System.Windows.Forms.Form' to type
'System.IConvertible'.
```

The catch clause can be used without arguments, in which case it catches any type of exception, and referred to as the general catch clause. It can also take an object argument derived from System.Exception, in which case it handles a specific exception. For example:

```
catch (InvalidCastException e)
{
}
```

It is possible to use more than one specific catch clause in the same try-catch statement. In this case, the order of the catch clauses is important because the catch clauses are examined in order. Catch the more specific exceptions before the less specific ones.

A throw statement can be used in the catch block to re-throw the exception, which has been caught by the catch statement. For example:

```
catch (InvalidCastException e)
{
    throw (e);     // Rethrowing exception e
}
```

If you want to re-throw the exception currently handled by a parameter-less catch clause, use the throw statement without arguments. For example:

```
catch
{
    throw;
}
```

Example

In this example, the try block contains a call to the method MyMethod() that may cause an exception. The catch clause contains the exception handler that simply displays a message on the screen. When the throw statement is called from inside MyMethod, the system looks for the catch statement and displays the message Exception caught.

```
using System;
function ProcessString(s)
    {
    if (s == null)
        {
        throw new ArgumentNullException( Argument is null );    "
    }
}

function fn()
    {
    try
        {
        s = null;
        ProcessString(s);
    }
    catch (Exception e)
        {
        print string.Format( {0} Exception caught. ," e.Message);          "
    }
}
```

Output

```
      fn()
Argument is null Exception caught.
```

try-finally (Visual APL Reference)

The finally block is useful for cleaning up any resources allocated in the try block as well as running any code that must execute even if there is an exception. Control is always passed to the finally block regardless of how the try block exits.

Remarks

Whereas catch is used to handle exceptions that occur in a statement block, finally is used to guarantee a statement block of code executes regardless of how the preceding try block is exited.

Example

In this example, there is one invalid conversion statement that causes an exception. When you run the program, you get a run-time error message, but the finally clause will still be executed and display the output.

```
// try-finally
using System;
function fn() {
    i = 1 2 3;
    try {
        // Invalid index
        a = i[5];
    } finally {
        print string.Format( i = {0} , i);          "       "
    }
}

      fn()
i =  System.Int32[]
Index was outside the bounds of the array.
```

Comments

Although an exception was caught, the output statement included in the finally block will still be executed.

Types (Visual APL Reference)

The Visual APL typing system contains the following categories:

Value types

Reference types

Variables of the value types store data, while those of the reference types store references to the actual data. Reference types are also referred to as objects.

Visual APL is ambivalent about data types, and all identifiers can be dynamically typed and contain any object or value type. Visual APL also supports strong typing within a class or function. Strong data typing is optional and is used primarily when there is a need to manipulate value types as scalars.

This section also introduces void.

Value types are also nullable, which means they can store an addition non-value state.

Value Types (Visual APL Reference)

ValueTypes are most often the primitive types used by the .Net framework, such as Int32, Int64, Double, Char, etc.

The value types consist of two main categories:

Structs

Enumerations

Structs fall into these categories:

Numeric types

Integral types

Floating-point types

Decimal

Boolean

Main Features of Value Types

Variables that are based on value types directly contain values. Assigning one value type variable to another copies the contained value. This differs from the assignment of reference type variables, which copies a reference to the object but not the object itself using the assign by reference operator =.

However, using the assign by value ← operator copies the value types out of a reference type and places them in a new instance of the reference type.

All value types are derived implicitly from the System.ValueType.

Unlike reference types, it is not possible to derive a new type from a value type. In particular this means that a class can not inherit from Int32, Double, etc.

Unlike reference types, it is not possible for a value type to contain the null value. However, the nullable types feature does allow values types to be assigned to null. In addition all arrays in Visual APL automatically promote to a nullable type when a null is assigned to a reference object.

Each value type has an implicit default constructor that initializes the default value of that type. For information on default values of value types, see Default Values Table.

Main Features of Simple Types

All of the simple types -- those integral to the Visual APL language -- are aliases of the .NET Framework System types. For example, int is an alias of System.Int32. For a complete list of aliases, see Built-In Types Table (Visual APL Reference).

Constant expressions, whose operands are all simple type constants, are evaluated at compilation time.

Simple types can be initialized using literals. For example, 'A' is a literal of the type char and 2001 is a literal of the type int.

Initializing Value Types

Local variables are created automatically when they are first assigned.

Example:

```
function fn() {
```

```
      a = 10
      b = 30.4
      c =  test                    "    "
}
```

Local variables in Visual APL are automatically initialized if declared using strong typing. Therefore, if you declare a local variable without initialization like this:

int myInt;

which is equivalent to:

```
myInt = 0;          // Assign an initial value, 0 in this example.
```

You can, of course, have the declaration and the initialization in the same statement like this:

```
int myInt = 0;
```

Once a variable has been declared as a specific type and shape it can only contain that type and shape.

Value Types Table (Visual APL Reference)

The following table lists the Visual APL value types by category.

Value type	Category
bool	Boolean
byte	Unsigned, numeric, integral
char	Unsigned, numeric, integral
decimal	Numeric, decimal
double	Numeric, floating-point
enum	Enumeration
float	Numeric, floating-point
int	Signed, numeric, integral
long	Signed, numeric, integral
sbyte	Signed, numeric, integral
short	Signed, numeric, integral
struct	User-defined structure
uint	Unsigned, numeric, integral
ulong	Unsigned, numeric, integral
ushort	Unsigned, numeric, integral

void (Visual APL Reference)

When used as the return type for a method, void specifies that the method does not return a value.

void is not allowed in a method's parameter list. A method that takes no parameters and returns no value is declared as follows. this is only valid within a class, and will not work in the session:

function void SampleMethod();

This will create a method signature which can be consumed by any .Net language and there will be no return from the method. However, when invoking the method in Visual APL, as with all methods that have a void return value, if an assignment is made from the method, the variable will receive a null.

void is an alias for the .NET Framework System.Void type.

Built-In Types Table (Visual APL Reference)

The following table shows the keywords for built-in Visual APL types, which are aliases of predefined types in the System namespace.

Visual APL Type .NET Framework Type

bool - System.Boolean

byte - System.Byte

sbyte - System.SByte

char - System.Char

decimal - System.Decimal

double - System.Double

float - System.Single

int - System.Int32

uint - System.UInt32

long - System.Int64

ulong - System.UInt64

object - System.Object

short - System.Int16

ushort - System.UInt16

string - System.String

ivar - APL variable type

Remarks

All of the types in the table, except ivar, object and string, are referred to as simple types.

The Visual APL type keywords and their aliases are interchangeable. For example, you can declare an integer variable by using either of the following declarations:

```
       int x = 123;
        System.Int32 x = 123;
       a = 123
       a.GetType()
System.Int32
```

To display the actual type for any Visual APL type, use the system method GetType(). For example, the following statement displays the system alias that represents the type of myVariable:

print myVariable.GetType();

You can also use the typeof operator.

Default Values Table (Visual APL Reference)

The following table shows the default values of value types returned by the default constructors. Default constructors are invoked by using the new operator, as follows:

```
int myInt = new int();
```

The preceding statement has the same effect as the following statement:

```
int myInt = 0;
```

Remember that variables are initialized when they are first assigned or when their type is defined.

Value type	Default value
bool	false
byte	0
char	'\0'
decimal	0.0M
double	0.0D
enum	The value produced by the expression (E)0, where E is the enum identifier.
float	0.0F
int	0
long	0L
sbyte	0
short	0
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to null.
uint	0
ulong	0
ushort	0

Explicit Numeric Conversions Table (Visual APL Reference)

Explicit numeric conversion is used to convert any numeric type to any other numeric type, for which there is no implicit conversion, by using a cast expression. The following table shows these conversions.

From	To
sbyte	byte, ushort, uint, ulong, or char
byte	Sbyte or char
short	sbyte, byte, ushort, uint, ulong, or char
ushort	sbyte, byte, short, or char
int	sbyte, byte, short, ushort, uint, ulong, or char
uint	sbyte, byte, short, ushort, int, or char
long	sbyte, byte, short, ushort, int, uint, ulong, or char
ulong	sbyte, byte, short, ushort, int, uint, long, or char
char	sbyte, byte, or short
float	sbyte, byte, short, ushort, int, uint, long, ulong, char, or decimal
double	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or decimal
decimal	sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or double

Remarks

The explicit numeric conversion may cause loss of precision or result in throwing exceptions.

When you convert a decimal value to an integral type, this value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination type, an OverflowException is thrown.

When you convert from a double or float value to an integral type, the value is truncated. If the resulting integral value is outside the range of the destination value, the result depends on the overflow checking context. In a checked context, an OverflowException is thrown, while in an unchecked context, the result is an unspecified value of the destination type.

When you convert double to float, the double value is rounded to the nearest float value. If the double value is too small or too large to fit into the destination type, the result will be zero or infinity.

When you convert float or double to decimal, the source value is converted to decimal representation and rounded to the nearest number after the 28th decimal place if required. Depending on the value of the source value, one of the following results may occur:

If the source value is too small to be represented as a decimal, the result becomes zero.

If the source value is NaN (not a number), infinity, or too large to be represented as a decimal, an OverflowException is thrown.

When you convert decimal to float or double, the decimal value is rounded to the nearest double or float value.

Floating-Point Types Table (Visual APL Reference)

The following table shows the precision and approximate ranges for the floating-point types.

Type	Approximate range	Precision
float	±1.5e−45 to ±3.4e38	7 digits
double	±5.0e−324 to ±1.7e308	15-16 digits

Implicit Numeric Conversions Table (Visual APL Reference)

The following table shows the predefined implicit numeric conversions. Implicit conversions might occur in many situations, including method invoking and assignment statements.

This is particularly important when selecting a method overload.

From	To
sbyte	short, int, long, float, double, or decimal
byte	short, ushort, int, uint, long, ulong, float, double, or decimal
short	int, long, float, double, or decimal
ushort	int, uint, long, ulong, float, double, or decimal
int	long, float, double, or decimal
uint	long, ulong, float, double, or decimal
long	float, double, or decimal
char	ushort, int, uint, long, ulong, float, double, or decimal
float	double
ulong	float, double, or decimal

Remarks

The conversions from int, uint, or long to float and from long to double may cause a loss of precision, but not a loss of magnitude.

There are no implicit conversions to the char type.

There are no implicit conversions between floating-point types and the decimal type.

A constant expression of type int can be converted to sbyte, byte, short, ushort, uint, or ulong, provided the value of the constant expression is within the range of the destination type.

In all cases, a conversion will be made if possible. This primarily occurs when using strong typed variables.

Example:

```
      int a = 10
      a = 33.4
      print a
33
```

When using dynamically typed variables this data conversion is not necessary.

Integral Types Table (Visual APL Reference)

The following table shows the sizes and ranges of the integral types, which constitute a subset of simple types.

Type	Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer
intn	-arbitrary large to arbitrary large	N bit integer

Modifiers (Visual APL Reference)

Modifiers are used to modify declarations of types and type members. This section introduces the Visual APL modifiers:

Modifier	Purpose
Access Modifiers public private internal protected	Specify the declared accessibility of types and type members.
abstract	Indicates that a class is intended only to be a base class of other classes.
event	Declares an event.
new	Hides an inherited member from a base class member.
override	Provides a new implementation of a virtual member inherited from a base class.
readonly	Declares a field that can only be assigned values as part of the declaration or in a constructor in the same class.
sealed	Specifies that a class cannot be inherited.
static	Declares a member that belongs to the type itself rather than to a specific object.
virtual	Declares a method or an accessor whose implementation can be changed by an overriding member in a derived class.
volatile	Indicates that a field can be modified in the program by something such as the operating system, the hardware, or a concurrently executing thread.

Types Reference Tables (Visual APL Reference)

The following reference tables summarize the Visual APL types:

Built-in Types Table

Integral types

Floating-point types

Default values

Value types

Implicit numeric conversions

Explicit Numeric Conversions Table

For information on formatting the output of numeric types, see □fmt.

Access Keywords (Visual APL Reference)

This section introduces the following access keywords:

```
base
```

Accesses the members of the base class.

```
this
```

Refers to the current instance of the class.

base (Visual APL Reference)

The base keyword is used to access members of the base class from within a derived class:

Call a method on the base class that has been overridden by another method.

Specify which base-class constructor should be called when creating instances of the derived class.

A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

It is an error to use the base keyword from within a static method.

Example

In this example, both the base class, Person, and the derived class, Employee, have a method named Getinfo. By using the base keyword, it is possible to call the Getinfo method on the base class, from within the derived class.

```
using System;
public class Person {
    protected string ssn =  444-55-6666 ;          "            "
    protected string name =  John L. Malgraine ;    "                  "

    public virtual void GetInfo() {
        print string.Format( Name: {0} , name);     "          "
        print string.Format( SSN: {0} , ssn);       "         "
    }
}
class Employee :  Person {
    public string id =  ABC567EFG ;            "          "
    public override void GetInfo() {
        // Calling the base class GetInfo method:
        base.GetInfo();
        print string.Format( Employee ID: {0} , id);"                 "
    }
}

class TestClass {
    public function fn() {
        Employee E = new Employee();
        E.GetInfo();
    }
}
```

This example shows how to specify the base-class constructor called when creating instances of a derived class.

```
using System;
public class BaseClass {
    int num;

    public BaseClass() {
        print  in BaseClass() ;       "              "
    }

    public BaseClass(int i) {
        num = i;
        Console.WriteLine( in BaseClass(int i) ); "                   "
    }

    public int GetNum() {
```

```
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base() {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i) {
    }

    public function fn()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
```

Output

```
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
```

Output

```
in BaseClass()
in BaseClass(int i)
```

this (Visual APL Reference)

The this keyword refers to the current instance of the class.

The following are common uses of this:

To qualify members hidden by similar names, for example:

```
public Employee(name, alias)
{
    this.name = name;
    this.alias = alias;
}
```

Literal Keywords (Visual APL Reference)

Visual APL has the following literal keywords:

```
null
true
false
default
```

null (Visual APL Reference)

The null keyword is a literal that represents a null reference, one that does not refer to any object. null is the default value of reference-type variables.

false Literal (Visual APL Reference)

Represents the boolean value true.

Example

```
using System;
    function fn() {
        a = false;
        if (a) {
           print  yes                    "    "
        } else {
           print  no                     "   "
        }
    }

Output
no
```

true Literal (Visual APL Reference)

Represents the boolean value true.

Example

```
using System;
    function fn() {
        a = true;
        if (a) {
            print  yes                    "    "
        }
    }

 Output
yes
```

default (Visual APL Reference)

The default keyword.

The default keyword can be used in the switch statement.

Contextual Keywords (Visual APL Reference)

A contextual keyword is used to provide a specific meaning in the code, but it is not a reserved word in Visual APL. The following contextual keywords are introduced in this section:

```
get   Defines an accessor method for a property or an indexer.
set   Defines an accessor method for a property or an indexer.
yield   Used in an iterator block to return a value to the enumerator object
or to signal the end of iteration.
value   Used to set accessors and to add or remove event handlers.
```

get (Visual APL Reference)

Defines an accessor method in a property or indexer that retrieves the value of the property or the indexer element. See Properties and Indexers for more information.

This is an example of a get accessor in a property called Seconds:

```
class TimePeriod
{
    private _seconds;
    public Seconds {
        get { return _seconds;  }
        set { _seconds = value; }
    }
}
```

set (Visual APL Reference)

Defines an accessor method in a property or indexer that retrieves the value of the property or the indexer element. See Properties and Indexers for more information.

This is an example of a set accessor in a property called Seconds:

```
class TimePeriod
{
    private _seconds;
    public Seconds {
        get { return _seconds;  }
        set { _seconds = value; }
    }
}
```

value (Visual APL Reference)

The implicit parameter value is used in setting accessors.

```
class TimePeriod
{
    private _seconds;
    public Seconds {
        get { return _seconds;  }
        set { _seconds = value; }
    }
}
```

yield (Visual APL Reference)

The yield keyword acts like return, but instead of exiting the function, the information is returned and when the function is accessed again, the function begins execution immediately following the yield.

See the tutorial for examples of using yield.

Namespace Keywords (Visual APL Reference)

This section describes the keywords and operators that are associated with using namespaces:

```
namespace
using
refbyfile
refbyname
```

using (Visual APL Reference)

The using keyword has two major uses:

As a directive, when it is used to create an alias for a namespace or to import types defined in other namespaces. See using Directive.

using Directive (Visual APL Reference)

The using directive has two uses:

To permit the use of types in a namespace so you do not have to qualify the use of a type in that namespace:

```
using System.Text;
```

To create an alias for a namespace or a type.

```
using Project = PC.MyCompany.Project;
```

The using keyword is also be used to create using statements, which define when an object will be disposed. See using Statement for more information.

Remarks

The scope of a using directive is limited to the file in which it appears.

Create a using alias to make it easier to qualify an identifier to a namespace or type.

Create a using directive to use the types in a namespace without having to specify the namespace. A using directive does not give you access to any namespaces that are nested in the namespace you specify.

The using directive provides the ability to include static methods on a type without having to specify the type.

Static methods with the AplFunctionAttribute are included as APL Functions, other methods appear as normal methods.

It is also possible to override primitives and system functions when the using directive specifies a type. Static methods on the type which have the AplOpsFunctionAttribute are evaluated and the returned hashtable is incorporated into the system primitives. When a primitive or system function name already exists, the current specification replaces the existing version. This makes it possible to override both primitives and system functions.

namespace (Visual APL Reference)

The namespace keyword is used to declare a scope. This namespace scope lets you organize code and gives you a way to create globally unique types.

```
namespace SampleNamespace
{
    class SampleClass{}
    interface SampleInterface{}
    enum SampleEnum{a,b}
}
```

Remarks

Within a namespace, you can declare one or more of the following types:

```
class
interface
enum
```

Whether or not you explicitly declare a namespace in a Visual APL source file, the compiler adds a default namespace. This unnamed namespace, sometimes called the global namespace, is present in every file. Any identifier in the global namespace is available for use in a named namespace.

Namespaces implicitly have public access and this is not modifiable.

prestmt (Visual APL Reference)

The prestmt directive allows code to be run during the directive processing state of the assembly creation process, and also during the initial startup of any classes present in the same file as the prestmt directive.

This directive effectively allows the specified code to run at the earliest possible moment during the building of an assembly, and also at the earliest possible moment during the instantiation of any static or instance classes in the same file as the prestmt directive.

Here is an example of using the prestmt directive:

```
using System
using System.IO
prestmt var1 = @ c:\clients\aplnext\    "                    "
refbyfile var1+ nsref1.dll             "          "
using nsref1.cs

namespace nsref2 {
 public class cs {
     public fn(a,b) {
         ⎕←var1+@ nsref1.dll            "          "
         return a add b
     }
 }
}
```

Here is the code for the "nsref1.dll" assembly referenced above:

```
using System
namespace nsref1 {
   public class cs {
       public ∇r←a add b {
           r←a + b
       }
   }
}
```

If we run this:

```
     using nsref2
   a = cs()
   a.fn(10,20)
c:\clients\aplnext\nsref1.dll
30
```

Notice that this brings in the nsref1.dll from the "c:\clients\aplnext\" directory by assigning this directory to the var1 variable.

This allows you to actually add code into the directive statements. The important thing to remember is that since this code is part of the directives, it is run at the time the assembly is being created and also when the assembly is being instanced or first accessed by .Net in the case of static assemblies.

This means that what is available when a dll is being instanced or used by another dll may not be available when the dll is being created.

An example of this would be referencing a *svglobal* variable. When the dll is being created, this would quite reasonably be *null*, however, when the dll is being instanced or referenced from another assembly it could have a value.

The prestmt permits any valid statement as its argument. So, you could do an *if* statement or even create a function. For instance:

```
prestmt svglobal svgv
prestmt if (svgv == null) {svgv = @c:\mydir\when\creating\dll\}
```

Notice that only one statement can be used as the argument to the prestmt directive, therefore statement separators like diamond would be an error.

This directive gives you very finite control over both the assembly build and the assembly initialization. However, because this occurs both when the assembly is being created and also at the time the class is being referenced, any errors in this section will result in creation errors or instantiation errors when you try to reference the class.

Since it is so far up in the creation process, you need to be careful about what you enter, as errors will keep the class from initializing or even being used.

refbyfile Directive (Visual APL Reference)

This adds a reference to a specific assembly referenced by file.

```
using System
refbyname @ c: \mydir\myfile.dll    "                    "
using myassembly
```

The namespace, myassembly, from the myfile.dll is referenced by the using.

refbyname Directive (Visual APL Reference)

This adds a reference to the assembly to the project.

For instance, the System.Windows.Forms assembly is not part of the default System assembly. So, to create a project which can access the windows forms the top of the file should include:

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

Operator Keywords (Visual APL Reference)

Used to perform miscellaneous actions such as creating objects, checking the run-time type of an object, obtaining the size of a type, and so forth. This section introduces the following keywords:

```
as   Converts an object to a compatible type.
is   Checks the run-time type of an object.
new
new Operator   Creates objects.
new Modifier   Hides an inherited member.
new Constraint   Qualifies a type parameter.
typeof   Obtains the System.Type object for a type.
true Literal   Represents the boolean value true.
false Literal   Represents the boolean value false.
```

as (Visual APL Reference)

Used to perform conversions between compatible reference types.

For example:

```
s = someObject as string;
if (s != null) {

    // someObject is a string.
}
```

is (Visual APL Reference)

Checks if an object is compatible with a given type. For example, it can be determined if an object is compatible with the string type like this:

```
if (obj is string) {
}
```

new (Visual APL Reference)

In Visual APL, the new keyword can be used as an operator.

It is specifically used for the creation of types, in the case of generics it makes the < and > delimiters.

Example:

```
a = new Dictionary<string, int>()
```

typeof (Visual APL Reference)

Used to obtain the System.Type object for a type. A typeof expression takes the following form:

```
type = typeof(int);
```

Method Parameters (Visual APL Reference)

If a parameter is declared for a method without ref or out, the parameter can have a value associated with it. That value can be changed in the method, but the changed value will not be retained when control passes back to the calling procedure. By using a method parameter keyword, you can change this behavior.

This section describes the keywords you can use when declaring method parameters:

```
params
ref
```

ref (Visual APL Reference)

The ref keyword causes arguments to be passed by reference. The effect is that any changes made to the parameter in the method will be reflected in that variable when control passes back to the calling method. "In Visual APL the ref key word is not required when calling a method, however, in other .Net languages, both the method definition and the calling method must explicitly use the ref keyword.

For example:

```
class RefExample
{
    static void Method(ref int i)
    {
        i = 44;
    }
}

The value of i will be 44 when the Method returns.

An argument passed to a ref parameter must first be initialized.

Using ref should only be used when creating methods within a formal class and
this is not intended for use in scripting.
```

params (Visual APL Reference)

The params keyword lets you specify a method parameter that takes an argument where the number of arguments is variable.

No additional parameters are permitted after the params keyword in a method declaration, and only one params keyword is permitted in a method declaration.

Note
This only works in a class and will not work in scripting scenarios. The params requires a type be specified for the argument, and in most cases this construct should be used in the creation of a Type to be formally consumed through an assembly.

Example:

```
public class cs1 {
    public function fn(params int[] a) {
        for (i = 0;i<a.Length;i++) {
            print a[i]
        }
    }
}


      a = cs1()
      a.fn(1,2,3)
1
2
3
```

Assembly Registration Tool (Regasm.exe)

This is the .Net Assembly Tool that makes it possible to register Assemblies you create which are to be exposed as COM objects. The Assembly Registration tool reads the metadata within an assembly and adds the necessary entries to the registry, which allows COM clients to create .NET Framework classes transparently. Once a class is registered, any COM client can use it as though the class were a COM class. The class is registered only once, when the assembly is installed. Instances of classes within the assembly cannot be created from COM until they are actually registered.

Microsoft has extensive documentation on using their numerous .Net Frameword Tools at: http://msdn2.microsoft.com/en-us/library/ms299153.aspx

```
regasm assemblyFile [options]
```

Parameters

Parameter	Description
assemblyFile	The assembly to be registered with COM.

Option	Description
/codebase	Creates a Codebase entry in the registry. The Codebase entry specifies the file path for an assembly that is not installed in the global assembly cache. You should not specify this option if you will subsequently install the assembly that you are registering into the global assembly cache. The *assemblyFile* argument that you specify with the **/codebase** option must be a strong-named assembly.
/registered	Specifies that this tool will only refer to type libraries that have already been registered.
/asmpath:directory	Specifies a directory containing assembly references. Must be used with the **/regfile** option.
/nologo	Suppresses the Microsoft startup banner display.
/regfile [**:***regFile*]	Generates the specified .reg file for the assembly, which contains the needed registry entries. Specifying this option does not change the registry. You cannot use this option with the **/u** or **/tlb** options.
/silent or **/s**	Suppresses the display of success messages.
/tlb [**:***typeLibFile*]	Generates a type library from the specified assembly containing definitions of the accessible types defined within the assembly.
/unregister or **/u**	Unregisters the creatable classes found in *assemblyFile*. Omitting this option causes Regasm.exe to register the creatable classes in the assembly.
/verbose	Specifies verbose mode; displays a list of any referenced assemblies for which a type library needs to be generated, when specified with the **/tlb** option.
/? or **/help**	Displays command syntax and options for the tool.

Note
The Regasm.exe command-line options are case insensitive. You only need to provide enough of the option to uniquely identify it. For example, **/n** is equivalent to **/nologo** and **/t:***outfile.tlb* is equivalent to **/tlb:** *outfile.tlb*.

Remarks

You can use the **/regfile** option to generate a .reg file that contains the registry entries instead of making the changes directly to the registry. You can update the registry on a computer by importing the .reg file with the Registry Editor tool (Regedit.exe). Note that the .reg file does not contain any registry updates that can be made by user-defined register functions. Note that the **/regfile** option only emits registry entries for managed classes. This option does not emit entries for **TypeLibID**s or **InterfaceID**s.

When you specify the **/tlb** option, Regasm.exe generates and registers a type library describing the types found in the assembly. Regasm.exe places the generated type libraries in the current working directory or the directory specified for the output file. Generating a type library for an assembly that references other assemblies may cause several type libraries to be generated at once. You can use the type library to provide type information to development tools like Visual Studio 2005. You should not use the **/tlb** option if the assembly you are registering was produced by the Type Library Importer (Tlbimp.exe). You cannot export a type library from an assembly that was imported from a type library. Using the **/tlb** option has the same effect as using the Type Library Exporter (Tlbexp.exe) and Regasm.exe, with the exception that Tlbexp.exe does not register the type library it produces. If you use the **/tlb** option to registered a type library, you can use **/tlb** option with the **/unregister** option to unregistered the type library. Using the two options together will unregister the type library and interface entries, which can clean the registry considerably.

When you register an assembly for use by COM, Regasm.exe adds entries to the registry on the local computer. More specifically, it creates version-dependent registry keys that allow multiple versions of the same assembly to run side by side on a computer. The first time an assembly is registered, one top-level key is created for the assembly and a unique subkey is created for the specific version. Each time you register a new version of the assembly, Regasm.exe creates a subkey for the new version.

For example, consider a scenario where you register the managed component, myComp.dll, version 1.0.0.0 for use by COM. Later, you register myComp.dll, version 2.0.0.0. You determine that all COM client applications on the computer are using myComp.dll version 2.0.0.0 and you decide to unregister myComponent.dll version 1.0.0.0. This registry scheme allows you to unregister myComp.dll version 1.0.0.0 because only the version 1.0.0.0 subkey is removed.

After registering an assembly using Regasm.exe, you can install it in the global assembly cache so that it can be activated from any COM client. If the assembly is only going to be activated by a single application, you can place it in that application's directory.

Examples

The following command registers all public classes contained in `myTest.dll`.

```
regasm myTest.dll
```

The following command generates the file `myTest.reg`, which contains all the necessary registry entries. This command does not update the registry.

```
regasm myTest.dll /regfile:myTest.reg
```

The following command registers all public classes contained in `myTest.dll`, and generates and registers the type library `myTest.tlb`, which contains definitions of all the public types defined in `myTest.dll`.

```
regasm myTest.dll /tlb:myTest.tlb
```

Arrays (Visual APL Programming Guide)

An array is a data structure that contains a number of variables of the same type. Arrays are declared with a type:

```
type[] arrayName;
```

The following examples create single-dimensional, multidimensional, and jagged arrays:

Visual APL

```
public class TestArraysClass
{
    public void StrongArrays()
    {
        // Declare a single-dimensional array
        int[] array1 = new int[5];

        // Declare and set array element values
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values
        int[,] multiDimensionalArray2 = new int[,]{ { 1, 2, 3 }, { 4, 5, 6 }
};

        // Declare a jagged array
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array
        // structure
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
    public void DynamicArrays() {
        a1 = 1 2 3 4 5
        a1 = 3⍴100
        a1 = ⍳10
        a1 = 3 3⍴⍳9
        a1 = 3 3⍴1 2 3 4 5 6 7 8 9
        a1 = (1,2,3) (1,2,3)
    }
    public void DynamicStrongArrays() {
        // this creates a strong typed array containing
        // the elements 0 through 9
        int[] array1 = ⍳10
        // b is dynamically typed and at this moment
```

```
        // contains the same elements as array1
        b = array1
        // now b will hold a text string
        b = "test""
    }
}
```

Array Overview

An array has the following properties:

- An array can be Single-Dimensional, Multidimensional or Jagged.
- The default value of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to **null**.
- Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. Since this type implements IEnumerable and IEnumerable, you can use foreach iteration on all arrays in Visual APL.

Single-Dimensional Arrays (Visual APL Programming Guide)

You can declare an array of five integers as in the following example:

In all cases strong and dynamically typed objects can be assigned to each other.

Visual APL

```
// this is a very fast way to create an array of zeros

int[] array = new int[5];

a1 = 5⍴0

array = ⍳10
```

This array contains the elements from `array[0]` to `array[4]`. The new operator is used to create the array and initialize the array elements to their default values. In this example, all the array elements are initialized to zero.

An array that stores string elements can be declared in the same way. For example:

Visual APL

```
string[] stringArray = new string[6];



a1 = 6⍴⊂""
```

Array Initialization

It is possible to initialize an array upon declaration, in which case, the rank specifier is not needed because it is already supplied by the number of elements in the initialization list. For example:

Visual APL

```
int[] array1 = new int[5] { 1, 3, 5, 7, 9 };

int[] array1 = {1, 3, 5, 7, 9};

a1 = 1 3 5 7 9

array1 = 3⍴100
```

A string array can be initialized in the same way. The following is a declaration of a string array where each array element is initialized by a name of a day:

Visual APL

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

When you initialize an array upon declaration, it is possible to use the following shortcuts:

Visual APL

```
int[] array2 = { 1, 3, 5, 7, 9 };

string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

It is possible to declare an array variable without initialization, but you must use the **new** operator when you assign an array to this variable. For example:

Visual APL

```
int[] array3;
```

```
array3 = new int[] { 1, 3, 5, 7, 9 };   // OK

//array3 = {1, 3, 5, 7, 9};   // Error
```

Value Type and Reference Type Arrays

Consider the following array declaration:

```
Visual APL

SomeType[] array4 = new SomeType[10];

A1 = new SomeType[10]
```

The result of this statement depends on whether SomeType is a value type or a reference type. If it is a value type, the statement results in creating an array of 10 instances of the type SomeType. If SomeType is a reference type, the statement creates an array of 10 elements, each of which is initialized to a null reference.

For more information on value types and reference types, see Types (Visual APL Reference).

Using foreach with Arrays (Visual APL Programming Guide)

Visual APL also provides the foreach statement. This statement provides a simple, clean way to iterate through the elements of an array. For example, the following code creates an array called `numbers` and iterates through it with the **foreach** statement:

```
Visual APL

int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (i in numbers)
{
    print i;
}
foreach (i in ⍳10) {
    print i;
}
```

With multidimensional arrays, you can use the same method to iterate through the elements, for example:

```
Visual APL

int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
foreach (i in numbers2D)
{
    print  i;
}
foreach (i in 3 3⍴⍳9) {
    print i;
}
```

The output of this example is:

```
9 99 3 33 5 55
```

However, with multidimensional arrays, using a nested for loop gives you more control over the array elements.

Jagged Arrays (Visual APL Programming Guide)

A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is sometimes called an "array of arrays." The following examples show how to declare, initialize, and access jagged arrays.

These are also referred to as nested arrays, and with dynamic typing can contain a heterogeneous mix of data types.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

Visual APL
```
int[][] jaggedArray = new int[3][];
ja = (1 2 3) (1 2 3 4) (1 2 3)
```

Before you can use `jaggedArray`, its elements must be initialized. You can initialize the elements like this:

Visual APL
```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
ja[0] = ⊂1 2 3 4 5 6
```

Each of the elements is a single-dimensional array of integers. The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.

It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size. For example:

Visual APL
```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
jaggedArray[2] = new int[] { 11, 22 };
ja[0] = ⊂11 12 13
```

You can also initialize the array upon declaration like this:

Visual APL
```
    int[][] jaggedArray2 = new int[][] {{1,3,5,7,9},{0,2,4,6},{11,22}};
    ja2 = (1 3 5 7 9) (0 2 4 6) (11 12)
```

A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to **null**.

You can access individual array elements like these examples:

Visual APL
```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

It is possible to mix jagged and multidimensional arrays. The following is a declaration and initialization of a single-dimensional jagged array that contains two-dimensional array elements of different sizes:

Visual APL
```
int[][,] jaggedArray4 = new int[3][,];
jaggedArray4[0] = new int[,]{{1,2,3},{4,5,6},{7,8,9}}
```

You can access individual elements as shown in this example, which displays the value of the element `[1,0]` of the first array (value `5`):

```
Visual APL
print "value: ", jaggedArray4[0][1, 0];
```

The method `Length` returns the number of arrays contained in the jagged array. For example, assuming you have declared the previous array, this line:

```
Visual APL
print “length: “+jaggedArray4.Length;
```

will return a value of 3.

Example

This example builds an array whose elements are themselves arrays. Each one of the array elements has a different size.

```
Visual APL
publicclass ArrayTest
{
    public ArrayTest()
    {
        // Declare the array of two elements:
        int[][] arr = new int[2][];
        // Initialize the elements:
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };
        // Display the array elements:
        for (int i = 0; i < arr.Length; i++)
        {
            print "Element: ", i);
            for (int j = 0; j < arr[i].Length; j++)
            {
                print  arr[i][j];
            }
        }
    }
}
```

Output

```
Element(0): 1 3 5 7 9

Element(1): 2 4 6 8
```

Multidimensional Arrays (Visual APL Programming Guide)

Arrays can have more than one dimension. For example, the following declaration creates a two-dimensional array of four rows and two columns:

```
Visual APL
int[,] array = new int[4, 2];
a1 ← 4 2⍴0
array ← 4 2⍴⍳8
```

Also, the following declaration creates an array of three dimensions, 4, 2, and 3:

```
Visual APL
int[, ,] array1 = new int[4, 2, 3];
a1 ← 4 2 3⍴0
array1 ← 4 2 3⍴0
```

Array Initialization

You can initialize the array upon declaration as shown in the following example:

```
Visual APL
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
int[, ,] array3D ← 3 3 3⍴⍳27
a3d ← 3 3 3⍴⍳29
```

You can also initialize the array without specifying the rank:

```
Visual APL
int[,] array4 ← 3 2⍴⍳6
a4 ← 3 2⍴⍳6
```

If you choose to declare an array variable without initialization, you must use the **new** operator to assign an array to the variable. For example:

```
Visual APL
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
a5 ← 3 3⍴⍳9
array5 ← 4 4⍴⍳16
```

It is important to remember, that when you indexing strong typed object, you should use a strong typed index to assure that all of the code is strong typed. You can also assign a value to an array element, for example:

```
Visual APL
array5[2, 1] = 25;
a4[2;1] = 10;
a4[1 2;1 2] = 10
array5[1 2;1 2] = 25
```

The following code example initializes the array variables to default (except for jagged arrays):

```
Visual APL
int[,] array6 = new int[10, 10];
```

Arrays as Objects (Visual APL Programming Guide)

In Visual APL, arrays are actually objects, and not just addressable regions of contiguous memory as in C and C++. Array is the abstract base type of all array types. You can use the properties, and other class members, that **Array** has. An example of this would be using the Length property to get the length of an array. The following code assigns the length of the numbers array, which is 5, to a variable called lengthOfNumbers:

```
Visual APL
numbers = 1 2 3 4 5
lengthOfNumbers = numbers.Length;
```

The **System.Array** class provides many other useful methods and properties for sorting, searching, and copying arrays.

Example:
This example uses the Rank property to display the number of dimensions of an array.

```
Visual APL
public class TestArraysClass
{
    public TestArrayClass ()
    {
        // Declare and initialize an array:
        int[,] theArray = new int[5, 10];
        print "The array has “+theArray.Rank+” dimensions.";
        newArray = 5 10⍴0
        print “"The array has “+newArray.Rank+” dimensions.";
    }
}
```

Output

```
The array has 2 dimensions.

The array has 2 dimensions.
```

Passing Arrays as Parameters (Visual APL Programming Guide)

Arrays may be passed to methods as parameters. As arrays are reference types, the method can change the value of the elements.

Passing single-dimensional arrays as parameters
You can pass an initialized single-dimensional array to a method. For example:

```
Visual APL
PrintArray(theArray);
```

The method called in the line above could be defined as:

```
Visual APL
void PrintArray(int[] arr)
{
    // method code
}
// or dynamic, no type given, will accept all types
void PrintArray(arr)
{
    // method code
}
```

You can also initialize and pass a new array in one step. For example:

```
Visual APL
PrintArray(new int[] { 1, 3, 5, 7, 9 });
// or dynamic
PrintArry(1 3 5 7 9)
```

Example 1

In the following example, a string array is initialized and passed as a parameter to the `PrintArray` method, where its elements are displayed:

```
Visual APL
Public class ArrayClass
{
    public void PrintArray(string[] arr)
    {
        for (int i = 0; i < arr.Length; i++)
        {
            print arr[i]
        }
    }
    public ArrayClass()
    {
        // Declare and initialize an array:
        string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu",
"Fri", "Sat" };
        // Pass the array as a parameter:
        PrintArray(weekDays);
    }
}
// or dynamic, in which case arr can be an array of anything
Public class ArrayClass
{
    public void PrintArray(arr)
```

```
    {
        for (i = 0; i < arr.Length; i++)
        {
            print arr[i]
        }
    }
    public ArrayClass()
    {
        // Declare and initialize an array:
        weekDays = "Sun"  "Mon"  "Tue"  "Wed"  "Thu"  "Fri"  "Sat"
        // Pass the array as a parameter:
        PrintArray(weekDays);
    }
}
```

Output 1

```
Sun Mon Tue Wed Thu Fri Sat
```

Passing multidimensional arrays as parameters

You can pass an initialized multidimensional array to a method. For example, if `theArray` is a two dimensional array:

Visual APL

```
PrintArray(theArray);
```

The method called in the line above could be defined as:

Visual APL

```
void PrintArray(int[,] arr)
{
    // method code
}
// or dynamic
void PrintArray(arr)
{
    // method code
}
```

You can also initialize and pass a new array in one step. For example:

Visual APL

```
PrintArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
// or dynamic
PrintArray(4 2⍴1+⍳8)
```

Example 2

In this example, a two-dimensional array is initialized and passed to the `PrintArray` method, where its elements are displayed.

Visual APL

```
public class ArrayClass2D
{
    static void PrintArray(int[,] arr)
    {
        // Display the array elements:
        for (int i = 0; i < 4; i++)
```

```
        {
            for (int j = 0; j < 2; j++)
         {
                print "Element(" + i + “,”+ j +”)”+ “=” +arr[i, j];
            }
        }
    }
    public ArrayClass2D()
    {
        // Pass the array as a parameter:
        PrintArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
    }
}
// or dynamic
public class ArrayClass2D
{
    static void PrintArray(arr)
    {
        // Display the array elements:
        for (i = 0; i < 4; i++)
        {
            for (j = 0; j < 2; j++)
            {
                print "Element(" + i + “,”+ j +”)”+ “=” +arr[i, j];
            }
        }
    }
    public ArrayClass2D()
    {
        // Pass the array as a parameter:
        PrintArray(4 2⍴1+⍳8);
    }
}
```

Output 2

```
Element(0,0)=1

Element(0,1)=2

Element(1,0)=3

Element(1,1)=4

Element(2,0)=5

Element(2,1)=6

Element(3,0)=7

Element(3,1)=8
```

How to: Implement Interface Events (Visual APL Programming Guide)

It is also possible for an interface to declare an event. This example demonstrates how to implement interface events in a class. Basically the rules are the same as when implementing any interface method or property.

To implement interface events in a class

- Declare the event in your class and then invoke it in the appropriate places.

```
public interface IDrawingObject
{
    event ShapeChanged;
}
public class MyEventArgs : EventArgs { }
public class Shape : IDrawingObject
{
    public event ShapeChanged;
    public void ChangeShape()
    {
        // Do something before the event…
        OnShapeChanged(new MyEventsArgs());
        // or do something after the event.
    }
    protected virtual void OnShapeChanged(MyEventArgs e)
    {
        if(ShapeChanged != null)
        {
           ShapeChanged(this, e);
        }
    }
}
```

How to: Raise Base Class Events in Derived Classes (Visual APL Programming Guide)

This simple example shows how to declare events in a base class so that they can also be raised from derived classes. This pattern is used extensively in Windows Forms classes in the .NET Framework base class library.

When you create a class that can be used as a base class for other classes, you address the fact that events are a special type of delegate that can only be invoked from within the class that declared them. Classes which are derived from or inherit from these classes cannot directly invoke events that are declared within the base class. Although sometimes you may want an event that can only be raised by the base class, in most cases you should enable the derived class to invoke base class events. To do this, you can create a protected invoking method in the base class that wraps the event. By calling or overriding this invoking method, derived classes can invoke the event indirectly.

Example

Visual APL

```
namespace BaseClassEvents
{
    using System;
    using System.Collections.Generic;

    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        private newArea;

        public ShapeEventArgs(d)
        {
            newArea = d;
        }
        public NewArea
        {
            get { return newArea; }
        }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected area;

        public Area
        {
            get { return area; }
            set { area = value; }
        }
        // The event. Note that by using the generic EventHandler<T> event
type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;
```

```
        public abstract void Draw() { };

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Make a temporary copy of the event to avoid possibility
            // of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is
            // raised.
            handler = ShapeChanged;
            if (handler != null)
            {
                handler(this, e);
            }
        }
    }
    public class Circle : Shape
    {
        private radius;
        public Circle(d)
        {
            radius = d;
            area = 3.14 × radius;
        }
        public void Update(d)
        {
            radius = d;
            area = 3.14 × radius;
            OnShapeChanged(new ShapeEventArgs(area));
        }
        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }
        public override void Draw()
        {
            print "Drawing a circle";
        }
    }
    public class Rectangle : Shape
    {
        private length;
        private width;
        public Rectangle(length, width)
        {
            this.length = length;
            this.width = width;
            area = length × width;
        }
        public void Update(length, width)
```

```
        {
            this.length = length;
            this.width = width;
            area = length × width;
            OnShapeChanged(new ShapeEventArgs(area));
        }
        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any rectangle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }
        public override void Draw()
        {
            print "Drawing a rectangle";
        }
    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        _list = null;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }
        public void AddShape(Shape s)
        {
            _list.Add(s);
            // Subscribe to the base class event.
            s.ShapeChanged += HandleShapeChanged;
        }
        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(sender, ShapeEventArgs e)
        {
            Shape s = (Shape)sender;

            // Diagnostic message for demonstration purposes.
            print String.Format("Received event. Shape area is now {0}",
e.NewArea);

            // Redraw the shape here.
            s.Draw();
        }
    }

    class Test
    {
```

```
        static void Test()
        {
            //Create the event publishers and subscriber
            c1 = Circle(54);
            r1 = Rectangle(12, 9);
            sc = ShapeContainer();

            // Add the shapes to the container.
            sc.AddShape(c1);
            sc.AddShape(r1);

            // Cause some events to be raised.
            c1.Update(57);
            r1.Update(7, 7);
        }
    }
}
```

Output

```
Received event. Shape area is now 178.98
Drawing a circle
Received event. Shape area is now 49
Drawing a rectangle
```

How to: Setting an Event or Subscribe to and Unsubscribe from Events (Visual APL Programming Guide)

You can subscribe to an event that is published by another class when you want to write custom code that is called when that event is fired. For example, you could subscribe to a button's "click" event in order to make your application do something useful when a user clicks the button.

To subscribe to events by using the Visual Studio 2005 IDE

1. If the **Properties** window is not visible, in Design view, right-click on the form or control for which you want to create an event handler, and select **Properties**.
2. On top of the **Properties** window, click the **Events** icon.
3. Double-click the event that you want to create, for example the Load event.

   Visual Visual APL creates an empty event handler method and adds it to your code. Alternatively you can add the code manually in Code view. For example, the following lines of code declare an event handler method that will be called when the Form class raises the Load event.

```
Visual APL
private void Form1_Load(sender, event)
{
    // Add your form load event handling code here.
}
```

   The line of code that is needed to subscribe to the event is also automatically generated in the InitializeComponent method in the Form1.Designer.cs file in your project. It looks like this:

```
this.Load += this.Form1_Load
```

To subscribe to events programmatically

1. Define an event handler method whose signature matches the delegate signature for the event. For example, if the event is based on the EventHandler delegate type, then the following code represents the method stub:

```
   void HandleCustomEvent(sender, event)
   {
      // Do something useful here.
   }
```

2. Use the addition assignment operator (+=) to attach your event handler to the event. In the following example, assume that an object named publisher has an event named RaiseCustomEvent. Note that the subscriber class needs a reference to the publisher class in order to subscribe to its events.

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

To subscribe to events by using an anonymous method

- Use the addition assignment operator (+=) to attach your method to the event. In the following example, assume that an object named publisher has an event named RaiseCustomEvent and that a CustomEventArgs class has also been defined to carry some kind of specialized event information. Note that the subscriber class needs a reference to publisher in order to subscribe to its events.

```
• □□□□□□□□□□□□□□□□□□□□□□□□□□publisher.RaiseCustomEvent += f(o, e)
•         {
•           print o
•           print e
```

```
        }
```

It is important to note that you cannot easily unsubscribe from an event if you used an anonymous method to subscribe to it. To unsubscribe in this scenario, go back to the code where you subscribe to the event, store the anonymous method in a variable, and then add the method to the event.

Unsubscribing

To prevent your event handler from being invoked when the event is fired, simply unsubscribe from the event. In order to prevent resource leaks, it is important to unsubscribe from events before you dispose of a subscriber object. Until you unsubscribe from an event, the multicast delegate that underlies the event in the publishing object has a reference the subscriber's event handler. As long as the publishing object holds that reference, your subscriber object will not be garbage collected.

To unsubscribe from an event

- Use the subtraction assignment operator (-=) to unsubscribe from an event:

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

When all subscribers have unsubscribed from an event, the event instance in the publisher class is set to null.

How to: Publish Events that Conform to .NET Framework Guidelines (Visual APL Programming Guide)

The following procedure demonstrates how to add events that follow the standard .NET Framework pattern to your own classes.

If you follow these steps you should have no problem generating events that can be consumed by any other .Net languages.

Although events in classes that you define can be based on any valid delegate type, including delegates that return a value, it is generally recommended that you base your events on the .NET Framework pattern by using **EventHandler**, as shown in the following example.

Take particular note that both the arguments to the CustomEventArgs class and the fields and properties in the class are strong typed. This is useful when the class is to be consumed by other .Net languages, such as C#. However, all of the typing could have been omitted, and the other .Net languages would have seen Object as the data types, instead of string.

To publish events based on the EventHandler pattern

1. (Skip this step and go directly to Step 3a if you have no need to send custom data with your event.)

   Declare your class at a scope visible to both your publisher and subscriber classes, and add the members needed to hold your custom event data. In this example, a simple string is returned.

```
public class CustomEventArgs : EventArgs
{
    public CustomEventArgs(string s)
    {
        msg = s;
    }
    private string msg;
    public string Message
    {
        get { return msg; }
    }
}
```

2. Declare the event in your publishing class by using one of the following steps.
   a. If you have no custom EventArgs class, then your Event type will be the non-generic EventHandler delegate. You do not need to declare it because it is already declared in the System namespace which is included by default in your Visual APL project:

```
public event RaiseCustomEvent;
```

   b. If you are using the generic version, you do not need a custom delegate. Instead, you specify your event type as EventHandler<CustomEventArgs>, substituting the name of your own class between the angle brackets.

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

Example

The following example demonstrates the steps given above using a custom EventArgs class and EventHandler<T> as the event type.

```
Visual APL
namespace DotNetEvents
{
    using System;
    using System.Collections.Generic;

    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
```

```
        public CustomEventArgs(string s)
        {
            message = s;
        }
        private string message;

        public string Message
        {
            get { return message; }
            set { message = value; }
        }
    }
    // Class that publishes an event
    class Publisher
    {
        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Did something"));
        }
        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
            // Make a temporary copy of the event to avoid possibility
            // of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is
            // raised.
               handler = RaiseCustomEvent;

            // Event will be null if there are no subscribers
            if (handler != null)
            {
                // Format the string to send inside the CustomEventArgs
               // parameter
                e.Message += String.Format(" at {0}",
                       DateTime.Now.ToString());

                // Use the () operator to raise the event.
                handler(this, e);
            }
        }
    }

    //Class that subscribes to an event
    class Subscriber
    {
        private id;
```

```
        public Subscriber(ID, pub)
        {
            id = ID;
            // Subscribe to the event using Visual APL 2.0 syntax
            pub.RaiseCustomEvent += HandleCustomEvent;
        }

        // Define what actions to take when the event is raised.
        void HandleCustomEvent(object sender, CustomEventArgs e)
        {
            print String.Format(id + " received this message: {0}",
e.Message);
        }
    }
    class Program
    {
        public Program()
        {
            pub = new Publisher();
            sub1 = new Subscriber("sub1", pub);
            sub2 = new Subscriber("sub2", pub);

            // Call the method that raises the event.
            pub.DoSomething();

            // Keep the console window open
            print "Press Enter to close this window.";
        }
    }
}
```

Main() Return Values (Visual APL Programming Guide)

The `Main` method can have a return type of **void**:

```
Visual APL

static void Main()

{

    //...

}
```

It can also return an **int**:

```
Visual APL

static int Main()

{

    //...

    return 0;

}
```

If the return value from `Main` is not to be used, then returning **void** allows slightly simpler code. However, returning an integer enables the program to relate status information to other programs or scripts that invoke the executable. An example of using the return value from `Main` is shown in the following example.

In this example a batch file is used to execute a program and test the return value of the `Main` function. When a program is executed in Windows, any value returned from the `Main` function is stored in an environment variable called `ERRORLEVEL`. By inspecting the `ERRORLEVEL` variable, batch files can therefore determine the outcome of execution. Traditionally, a return value of zero indicates successful execution. Below is a very simple program that returns zero from the `Main` function.

Visual APL

```
class MainReturnValTest

{

    static int Main()

    {

        //...

        return 0;

    }

}
```

Because this example uses a batch file, it is best to compile this code from the command line.

Next, a batch file is used to invoke the executable resulting from the previous code example. Because the code returns zero, the batch file will report success, but if the previous code is changed to return a non-zero value, and is then re-compiled, subsequent execution of the batch file will indicate failure.

```
rem test.bat

@echo off

MainReturnValueTest

@if "%ERRORLEVEL%" == "0" goto good

:fail

    echo Execution Failed

    echo return value = %ERRORLEVEL%
```

```
    goto end

:good

    echo Execution Succeded

    echo return value = %ERRORLEVEL%

    goto end

:end
```

```
Execution Succeded

return value = 0
```

How to: Display Command Line Arguments (Visual APL Programming Guide)

Arguments provided to an executable (exe) on the command-line are accessible through an optional parameter to `Main`. The arguments are provided in the form of an array of strings. Each element of the array contains one argument. White-space between arguments is removed. For example, consider these command-line invocations of a fictitious executable:

Input on Command-line	Array of strings passed to Main
exec.exe a b c	"a" "b" "c"
exec.exe one two	"one" "two"
exec.exe “one two” three	"one two" "three"

Example

This example displays the command line arguments passed to a command-line application. The output shown is for the first entry in the table above.

```
Visual APL
class CommandLine
{
    static void Main(string[] args)
    {
        // The Length property provides the number of array elements
        System.Console.WriteLine("parameter count = {0}", args.Length);
        for (int i = 0; i < args.Length; i++)
        {
            System.Console.WriteLine("Arg[{0}] = [{1}]", i, args[i]);
        }
    }
}
```

Output

```
Visual APL
parameter count = 3
Arg[0] = [a]
Arg[1] = [b]
Arg[2] = [c]
```

How to: Access Command-Line Arguments Using foreach (Visual APL Programming Guide)

Another approach to iterating over an array is to use the foreach statement as shown in this example. The **foreach** statement can be used to iterate over an array, a .NET Framework collection class, or any class or struct that implements the IEnumerable interface.

Example

This example demonstrates how to print out the command line arguments using **foreach**.

```
Visual APL
// arguments: John Paul Mary
class CommandLine2
{
   static void Main(string[] args)
    {
        System.Console.WriteLine("Number of command line parameters = {0}",
args.Length);

        foreach (s in args)
        {
            System.Console.WriteLine(s);
        }
    }
}
```

Output

```
Visual APL
Number of command line parameters = 3
John
Paul
Mary
```

Indexers (Visual APL Programming Guide)

Indexers make it possible for instances of a class to be indexed in the same way as arrays. Indexers are similar to properties except that their accessors take parameters.

In the following example, a class is defined and provided with simple get and set accessor methods as a means for assigning and retrieving values. The class `Test` creates an instance of this class for storing strings.

Visual APL

```
class SampleCollection
{
    private int arr = new int[100];
    public int this[int i]
    {
        get
        {
            return arr[i];
        }
        set
        {
            arr[i] = value;
        }
    }
}
// This class shows how client code uses the indexer
public class Test
{
    public  Test()
    {
        intCollection = new SampleCollection();
        stringCollection[0] = 1000;
        print intCollection[0];
    }
}
```

Indexers Overview

- Indexers enable objects to be indexed in a similar way to arrays.
- A **get** accessor returns a value. A **set** accessor assigns a value.
- The this keyword is used to define the indexers.
- The value keyword is used to define the value being assigned by the **set** indexer.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Comparison Between Properties and Indexers (Visual APL Programming Guide)

Indexers are similar to properties. Except for the differences shown in the following table, all of the rules defined for property accessors apply to indexer accessors as well.

Property	Indexer
Allows methods to be called as though they were public data members.	Allows methods on an object to be called as though the object is an array.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A get accessor of a property has no parameters.	A **get** accessor of an indexer has the same formal parameter list as the indexer.
A set accessor of a property contains the implicit **value** parameter.	A **set** accessor of an indexer has the same formal parameter list as the indexer, in addition to the value parameter.

Using Indexers (Visual APL Programming Guide)

Indexers allow you to index a class or interface in the same way as an array. For more information about using indexers with an interface, see Interface Indexers.

To declare an indexer on a class or struct, use the this keyword, as in this example:

```
public int this[int index]    // Indexer declaration
{
    // get and set accessors
}
```

Remarks

The type of an indexer and the type of its parameters must be at least as accessible as the indexer itself. For more information about accessibility levels, see Access Modifiers.

The signature of an indexer consists of the number and types of its defined parameters. It does not include the indexer type or the names of the defined parameters. If you declare more than one indexer in the same class, they must have different signatures.

An indexer value is not classified as a variable; therefore, it is not possible to pass an indexer value as a ref or out parameter.

The default name of the indexer is Item, however, substituting another name for this in the indexer declaration changes the name of the indexer to the name given.

Example 1

The following example shows how to declare a private array field, `arr`, and an indexer. Using the indexer enables direct access to the instance `test[i]`. The alternative to using the indexer is to declare the array as a public member and access its members, `arr[i]`, directly.

Visual APL

```
class IndexerClass
{
    private int[] arr = new int[100];
    public int this[int index]   // Indexer declaration
    {
        get
        {
            // Check the index limits.
            if (index < 0 || index >= 100)
            {
                return 0;
            }
            else
            {
                return arr[index];
            }
        }
        set
        {
            if (!(index < 0 || index >= 100))
            {
                arr[index] = value;
```

```
            }
        }
    }
}
class Test
{
    public Test()
    {
        test = new IndexerClass();
        // Call the indexer to initialize the elements #3 and #5.
        test[3] = 256;
        test[5] = 1024;
        for (int i = 0; i <= 10; i++)
        {
            print "Element “ + i + ” = " + test[i];
        }
    }
}
```

Output

```
Element #0 = 0

Element #1 = 0

Element #2 = 0

Element #3 = 256

Element #4 = 0

Element #5 = 1024

Element #6 = 0

Element #7 = 0

Element #8 = 0

Element #9 = 0

Element #10 = 0
```

Notice that when an indexer's access is evaluated, for example, in a **print** statement, the **get** accessor is invoked. Therefore, if no **get** accessor exists, a compile-time error occurs.

Indexing Using Other Values

Visual APL does not limit the index type to integer. For example, it may be useful to use a string with an indexer. Such an indexer might be implemented by searching for the string within the collection, and returning the appropriate value. As accessors can be overloaded, the string and integer versions can co-exist.

Example 2

In this example, a class is declared that stores the days of the week. A **get** accessor is declared that takes a string, the name of a day, and returns the corresponding integer. For example, Sunday will return 0, Monday will return 1, and so on.

Visual APL

```
// Using a string as an indexer value
```

```
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };
    // This method finds the day or returns -1
    private int GetDay(string testDay)
    {
        int i = 0;
        foreach (string day in days)
        {
            if (day == testDay)
            {
                return i;
            }
            i++;
        }
        return -1;
    }
    // The get accessor returns an integer for a given string
    public int this[string day]
    {
        get
        {
            return (GetDay(day));
        }
    }
}
public class Test
{
    public Test()
    {
        week = new DayCollection();
        print “value: “+week["Fri"];
        print “value: “+week["Made-up Day"];
    }
}
```

Output

```
5

-1
```

Robust Programming

There are two main ways in which the security and reliability of indexers can be improved:

- Always ensure that your code performs **range** checks when setting and retrieving values from any buffer or array accessed by the indexers.
- Always ensure that your code performs **type** checks when setting and retrieving values from any buffer or array accessed by the indexers.

Indexers in Interfaces (Visual APL Programming Guide)

Indexers can be declared on an interface (Visual APL Reference). Accessors of interface indexers differ from the accessors of class indexers in the following ways:

- An interface accessor does not have a body.

Thus, the purpose of the accessor is to indicate whether the indexer is read-write, read-only, or write-only.

The following is an example of an interface indexer accessor:

```
Visual APL
public interface ISomeInterface
{
    //...
    // Indexer declaration:
    public string this[int index]
    {
        get;
        set;
    }
}
```

The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

Example

The following example shows how to implement interface indexers.

```
Visual APL
// Indexer on an interface:
public interface ISomeInterface
{
    // Indexer declaration:
    public int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : ISomeInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        get
        {
            // Check the index limits.
            if (index < 0 || index >= 100)
            {
                return 0;
            }
            else
            {
```

```
                return arr[index];
            }
        }
        set
        {
            if (!(index < 0 || index >= 100))
            {
                arr[index] = value;
            }
        }
    }
}
Public class Test
{
    public  void Test()
    {
        test = new IndexerClass();
        // Call the indexer to initialize the elements #2 and #5.
        test[2] = 4;
        test[5] = 32;
        for (int i = 0; i <= 10; i++)
        {
            print "Element “+j+” = ", test[i];
        }
    }
}
```

Output

```
Element #0 = 0

Element #1 = 0

Element #2 = 4

Element #3 = 0

Element #4 = 0

Element #5 = 32

Element #6 = 0

Element #7 = 0

Element #8 = 0

Element #9 = 0

Element #10 = 0
```

How to: Create and Terminate Threads (Visual APL Programming Guide)

This example demonstrates how an auxiliary or worker thread can be created and used to perform processing in parallel with the primary thread. Making one thread wait for another and gracefully terminating a thread are also demonstrated. For background information on multi-threading, see Managed Threading and Using Threading (Visual APL Programming Guide).

The example creates a class named `Worker` that contains the method that the worker thread will execute called `DoWork`. The worker thread will begin execution by calling this method, and terminate automatically when this method returns. The `DoWork` method looks like this:

```
Visual APL
    public function DoWork() {
        a = Form()
        t = TextBox()
        t.Multiline = true
        t.Size = Size(200, 200)
        a.Controls.Add(t)
        a.Show()
        for (i=0;i<100;i++) {
            if (_shouldStop) {
                break;
            }
            t.AppendText(i+". worker thread: working...\n")
        }
        MessageBox.Show("Worker thread: terminating gracefully")
    }
```

The `Worker` class contains an additional method that is used to indicate to `DoWork` that it should return. This method is called `RequestStop`, and looks like this:

```
Visual APL
public void RequestStop()
{
    _shouldStop = true;
}
```

The `RequestStop` method merely assigns the `_shouldStop` data member to **true**. Because this data member is checked by the `DoWork` method, this has the indirect effect of causing `DoWork` to return, thereby terminating the worker thread. However, it is important to note that `DoWork` and `RequestStop` will be executed by different threads. `DoWork` is executed by the worker thread, and `RequestStop` is executed by the primary thread, so the `_shouldStop` data member is declared **volatile**, like this:

```
Visual APL
private volatile bool _shouldStop;
```

The **volatile** keyword alerts the compiler that multiple threads will access the `_shouldStop` data member, and therefore it should not make any optimization assumptions about the state of this member. For more information, see volatile (Visual APL Reference).

The use of **volatile** with the _shouldStop data member allows us to safely access this member from multiple threads without the use of formal thread synchronization techniques, but only because _shouldStop is a **bool**. This means that only single, atomic operations are necessary to modify _shouldStop. If, however, this data member were a class, struct, or array, accessing it from multiple threads would likely result in intermittent data corruption. Consider a thread that changes the values in an array. Windows regularly interrupts threads in order to allow other threads to execute, so this thread could be halted after assigning some array elements but before assigning others. This means the array now has a state that the programmer never intended, and another thread reading this array may fail as a result.

Before actually creating the worker thread, the test function creates a Worker object and an instance of Thread. The thread object is configured to use the Worker.DoWork method as an entry point by passing a reference to this method to the **Thread** constructor, like this:

```
Visual APL
wo = Worker();
wt = new Thread((ThreadStart)wo.DoWork);
```

At this point, although the worker thread object exists and is configured, the actual worker thread has yet been created. This does not happen until test calls the Start method:

```
Visual APL
wt.Start();
```

At this point the system initiates the execution of the worker thread, but it does so asynchronously to the primary thread. This means that the test function continues to execute code immediately while the worker thread simultaneously undergoes initialization. To insure that the test function does not try to terminate the worker thread before it has a chance to execute, the test function loops until the worker thread object's IsAlive property gets set to **true**:

```
Visual APL
while (!wt.IsAlive);
```

Next, the primary thread is halted briefly with a call to Sleep. This insures that the worker thread's DoWork function will execute the loop inside the DoWork method for a few iterations before the test function executes any more commands:

```
Visual APL
Thread.Sleep(300);
```

After the 300 millisecond elapses, test signals to the worker thread object that it should terminate using the Worker.RequestStop method introduced previously:

```
Visual APL
wo.RequestStop();
```

It is also possible to terminate a thread from another thread with a call to Abort, but this forcefully terminates the affected thread without concern for whether it has completed its task and provides no opportunity for the cleanup of resources. The technique shown in this example is preferred.

Finally, the test function calls the Join method on the worker thread object. This method causes the current thread to block, or wait, until the thread that the object represents terminates. Therefore **Join** will not return until the worker thread returns, thereby terminating itself:

```
Visual APL
wt.Join();
```

At this point only the primary thread executing `test` exists. It displays one final message, and then returns, terminating the primary thread as well.

The complete example appears below.

Example

Visual APL

```
using System
using System.Threading
refbyname System.Windows.Forms
using System.Windows.Forms
refbyname System.Drawing
using System.Drawing


public class Worker {

    // This method will be called when the thread is started.
    public function DoWork() {

        a = Form()
        t = TextBox()
        t.Multiline = true
        t.Size = Size(200, 200)
        a.Controls.Add(t)
        a.Show()
        for (i=0;i<100;i++) {
            if (_shouldStop) {
                break;
            }
            t.AppendText(i+". worker thread: working...\n")
        }
        MessageBox.Show("Worker thread: terminating gracefully")
    }
    public void RequestStop() {
        _shouldStop = true;
    }

    // Volatile is used as hint to the compiler that this data
    // member will be accessed by multiple threads.
    private volatile _shouldStop = false;
}
static function test() {

    // Create the thread object. This does not start the thread.
    wo = Worker()
    wt = Thread((ThreadStart)wo.DoWork)

    // Start the worker thread.
    wt.Start()
    print "main thread: starting worker thread..."

    // Put the main thread to sleep for 300 milliseconds to
```

```
    // allow the worker thread to do some work:
    Thread.Sleep(300)

    // Request that the worker thread stop itself:
    wo.RequestStop()

    // Use the Join method to block the current thread
    // until the object's thread terminates.
    wt.Join()
    print "main thread: worker thread has terminated"
}
```

Sample Output

```
main thread: starting worker thread...
1. worker thread: working...
2. worker thread: working...
3. worker thread: working...
4. worker thread: working...
5. worker thread: working...
6. worker thread: working...
7. worker thread: working...
8. worker thread: working...
9. worker thread: working...
10. worker thread: working...
11. worker thread: working...
Shown in MessageBox -  worker thread: terminating gracefully...
main thread: worker thread has terminated
```

How to: Access a Collection Class with foreach (Visual APL Programming Guide)

The following code sample illustrates how to write a non-generic collection class that can be used with **foreach**. The class is a string tokenizer, similar to the C run-time function **strtok**.

In the following example, `Tokens` breaks the sentence "This is a sample sentence." into tokens using ' ' and '-' as separators, and enumerates those tokens with the **foreach** statement:

```
Visual APL
f = Tokens("This is a sample sentence.", (' ' '-'));
foreach (item in f)
{
    print item;
}
```

Internally, `Tokens` uses an array, which implements `IEnumerator` and `IEnumerable` itself. The code example could have used the array's enumeration methods as its own, but that would have defeated the purpose of this example.

In Visual APL, it is not strictly necessary for a collection class to inherit from `IEnumerable` and `IEnumerator` in order to be compatible with **foreach**; as long as the class has the required `GetEnumerator`, `MoveNext`, `Reset`, and `Current` members, it will work with **foreach**. Omitting the interfaces has the advantage of allowing you to define the return type of `Current` to be more specific than **object**, thereby providing type-safety.

The disadvantage of omitting **IEnumerable** and **IEnumerator** is that the collection class is no longer interoperable with the **foreach** statements, or equivalents, of other common language runtime-compatible languages.

Creating CLS compliant enumerators within Visual APL for interoperability with other common language runtime-compatible languages, by inheriting from **IEnumerable** and **IEnumerator** and using explicit interface implementation is demonstrated in the following example.

Example

```
Visual APL
using System
using System.Collections;

// Declare the Tokens class:
public class Tokens : IEnumerable
{
    private elements;

    public Tokens(source, delimiters)
    {
        // Parse the string into tokens:
        elements = source.Split(delimiters, source.Length);
    }

    // IEnumerable Interface Implementation:
    //   Declaration of the GetEnumerator() method
```

```
    //   required by IEnumerable
    //   the return type of IEnumerator is required for override
    public IEnumerator GetEnumerator()
    {
        return TokenEnumerator(this);
    }
    // Inner class implements IEnumerator interface:
    private class TokenEnumerator : IEnumerator
    {
        private position = -1;
        private t;

        public TokenEnumerator(t)
        {
            this.t = t;
        }
        // Declare the MoveNext method required by IEnumerator:
        // note that the return type must be bool to override the MoveNext
method on the
        // interface IEnumerator
        public bool MoveNext()
        {
            if (position < t.elements.Length - 1)
            {
                position++;
                return true;
            }
            else
            {
                return false;
            }
        }
        // Declare the Reset method required by IEnumerator:
        // Reset also requires void as the return type for override
        public void Reset()
        {
            position = -1;
        }
        // Declare the Current property required by IEnumerator:
        // Current returns type, so there is no need to specify the return
type for override
        public Current
        {
            get
            {
                return t.elements[position];
            }
        }
    }
    // Test Tokens, TokenEnumerator
    public static test()
    {
        // Testing Tokens by breaking the string into tokens:
        f = new Tokens("This is a sample sentence.", (' ' '-'));
```

```
        foreach (item in f)
        {
            print item;
        }
    }
}
```

Output

```
This
is
a
sample
sentence.
```

How to: Use COM Interop to Create an Excel Spreadsheet (Visual APL Programming Guide)

The following code example illustrates how to use **COM interop** to create an **Excel** spreadsheet. For more information on **Excel**, see Microsoft Excel Objects, and Open Method

This example illustrates how to open an existing **Excel** spreadsheet in Visual APL using .NET Framework **COM interop** capability. The **Excel** assembly is used to open and enter data into a range of cells in the **Excel** spreadsheet.

Note

You must have **Excel** installed on your system for this code to run properly.

Note

The members, such as properties, methods, etc displayed with intellisense may differ from those available on the object. Simply type the member you would normally select from intellisense and the code should run fine.

To create an Excel spreadsheet with COM interop

Visual APL

```
using System;
refbyname Microsoft.Office.Interop.Excel
using Microsoft.Office.Interop;
using Microsoft.Office.Interop.Excel;
    function excel()
    {
        xlApp = Excel.ApplicationClass();
        if (xlApp == null)
        {
            print "EXCEL could not be started. Check that your office
installation and project references are correct.";
            return;
        }
        xlApp.Visible = true;
        wb = xlApp.Workbooks.Add(XlWBATemplate.xlWBATWorksheet);
        ws = (Worksheet)wb.Worksheets[1];
        if (ws == null)
        {
            print "Worksheet could not be created. Check that your office
installation and project references are correct.";
        }
        // Select the Excel cells, in the range c1 to c7 in the worksheet.
        aRange = ws.get_Range("C1", "C7");
        if (aRange == null)
        {
            print "Could not get a range. Check to be sure you have the
correct versions of the office DLLs.";
        }
        // Change the cells in the C1 to C7 range of the worksheet to the
number 8.
        aRange.Value2 = 8;
    }
```

Security

To use **COM interop**, you must have **administrator** or **Power User** security permissions. For more information on security, see .NET Framework Security.

How to: Write a Copy Constructor (Visual APL Programming Guide)

Visual APL does not include an explicit copy constructor, but Visual APL does permit multiple instance constructors based on signature. If you create an instance of a type and want to copy the values from an existing instantiation, you can write the appropriate constructor method.

Example

In this example, the **Employee** class contains a constructor that takes as the argument another object of type **Employee**. The contents of the fields in this object are then assigned to the fields in the new object.

Visual APL

```
class Employee
{
    private name;
    private age;

    // Copy constructor.
    public Employee(previousEmployee)
    {
        name = previousEmployee.name;
        age = previousEmployee.age;
    }

    // Instance constructor.
    public Employee(name, age)
    {
        this.name = name;
        this.age = age;
    }

    // Get accessor.
    public string Details
    {
        get
        {
            return name + " is " + age.ToString();
        }
    }
}
class TestEmployee
{
    static void Main()
    {
        // Create a new person object.
        person1 = Employee("Sam", 40);

        // Create another new object, copying person1.
        person2 = Employee(person1);
        print person2.Details;
    }
}
```

Output

```
Sam is 40
```

How to: Implement Interface Members (Visual APL Programming Guide)

This example creates an interface, `IDimensions`, and a class, `Box`, which explicitly implements or inherits from the interface members `getLength` and `getWidth`. The members are accessed through the interface instance `dimensions`.

Example

```
Visual APL
interface IDimensions
{
    function float getLength()
    function float getWidth()
}

class Box : IDimensions
{
    public float lengthInches;
    public float widthInches;
    Box(length, width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    public float getLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    public float IDimensions.getWidth()
    {
        return widthInches;
    }

    public void Test()
    {
        // Declare a class instance box1:
        box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        dimensions = (IDimensions)box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        // print "Length: " + box1.getlength();
        // print "Width: " + box1.getwidth());

        // Print out the dimensions of the box by calling the methods
```

```
        // from an instance of the interface:
        print "Length: " + dimensions.getLength();
        print "Width: " + dimensions.getWidth();
    }
}
```

Output

```
Length: 30
Width: 20
```

Static Constructors (Visual APL Programming Guide)

A static constructor is run only once when the system first loads a type, and is used to initialize any static data, or to perform a particular action that needs to be performed once only. The system automatically calls this method before the first instance is created or any static members are referenced.

```
Visual APL
class SimpleClass
{
    // Static constructor
    static SimpleClass()
    {
        //...
    }
}
```

Static constructors have the following properties:

- A static constructor does not take access modifiers or have parameters.
- A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
- A static constructor cannot be called directly.
- The user has no control on when the static constructor is executed in the program.
- A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
- Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the **LoadLibrary** method.

Example

In this example, the class One has a static constructor and one static member, Two(). When Two() is called, the static constructor is invoked to initialize the class.

```
Visual APL
public class One
{
    // Static constructor:
    static One()
    {
        print "The static constructor invoked."
    }
    public static void Two()
    {
        print "The Two method invoked."
    }
}
class TestOne
{
    static void Main()
    {
        One.Two();
    }
}
```

Output

```
The static constructor invoked.
```

The Drive method invoked.

Instance Constructors (Visual APL Programming Guide)

A class can have both static and instance members. An instance class can have multiple constructors or latent expressions. A constructor is the latent expression or method that is invoked when the class is first used, or instantiated. When a method is created with the same name as the class this becomes a constructor or latent expression. In addition to running the code in the constructor method, the constructor also assures that all of the fields (global variable) x and y in this case, are set to the original default values when the class is instantiated. There can be more than one constructor, as the entire signature of the constructor is used to determine which constructor is called, for example:

Visual APL
```
class Coordinates
{
    public x, y;

    // constructor
    public Coordinates()
    {
        x = 0;
        y = 0;
    }
}
```
In this case the method named Coordinates is called when ever an instance of the Coordinates class is created. When a constructor or latent expression has no arguments, then it is called the default constructor, which other programs will use when no arguments are available to create an instance of a class. However, there can be numerous latent expressions or constructor methods. For example, we can add a constructor to the `Coordinates` class that allows us to specify the initial values for the data members:

Visual APL
```
// A constructor with two arguments:
public Coordinates(x, y)
{
    this.x = x;
    this.y = y;
}
```
This constructor will assign values to the global variables, or fields, x and y when the class is initialized, such as:

Visual APL
```
p1 = new Coordinates();
p2 = new Coordinates(5, 3);
```
A class can inherit from another class. In this case, the constructors also call the instance constructors of base classes, that is the class from which they inherit. The constructor or latent expression automatically selects the base constructor that matches the signature, or uses the base default constructor if no match is found. The following show this:

Visual APL
```

class Circle : Shape
{
    public Circle(radius)
    {
    }
}
```

In this case the Circle constructor also calls the Shape constructor that matches the arguments.

Example 1

This example shows a class which has two constructors or latent expressions. They have the same name as the class, but are made unique by having different arguments.

Visual APL

```
class Coordinates
{
    public x, y;

    // Default constructor:
    public Coordinates()
    {
        x = 0;
        y = 0;
    }

    // A constructor with two arguments:
    public Coordinates(x, y)
    {
        this.x = x;
        this.y = y;
    }

    // Override the ToString method:
    public override string ToString()
    {
        return (System.String.Format("({0},{1})", x, y));
    }
}

class TestClass
{
    static void Test()
    {
        p1 = new Coordinates();
        p2 = new Coordinates(5, 3);

        // Display the results using the overriden ToString method:
        print "Coordinates #1 at "+ p1;
        print "Coordinates #2 at "+ p2;
    }
}
```

Output

```
Coordinates #1 at (0,0)

Coordinates #2 at (5,3)
```

Example 2

This class does not have an explicit constructor, but instead one is automatically created, which assures that the fields (global variables) age and name are set each time the class is instantiated.

Visual APL

```
public class Person
{
    public age = 0;
    public name = “”;
}

class TestPerson
{
    public static void Test()
    {
        p = new Person();

        print "Name: "+p.name+", Age: " + p.age;
    }
}
```

Output

```
Name: , Age: 0
```

Example 3

This example shows how a constructor will also call a base constructor, or the constructor of the class from which the present class inherited. The `Circle` class is derived from the general class `Shape`, and the `Cylinder` class is derived from the `Circle` class. The constructor on each derived class is using its base class initializer, constructor or latent expression.

Visual APL

```
abstract class Shape
{
    public pi = System.Math.PI;
    public x, y;

    public Shape(x, y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract Area();
}

class Circle : Shape
{
    public Circle(radius) : base(radius, 0)
    {
    }
    public override Area()
    {
        return pi × x × x;
    }
}
class Cylinder : Circle
```

```
{
    public Cylinder(radius, height) : base(radius)
    {
        y = height;
    }
    public override Area()
    {
        return (2 * base.Area()) + (2 * pi * x * y);
    }
}

class TestShapes
{
    public void TestShapes()
    {
        radius = 2.5;
        height = 3.0;

        ring = Circle(radius);
        tube = Cylinder(radius, height);

        print "Area of the circle = "+ ring.Area();
        print "Area of the cylinder = "+ tube.Area();
    }
}
```

Output

```
Area of the circle = 19.63

Area of the cylinder = 86.39
```

Example COM Class (Visual APL Programming Guide)

The following is an example of a class that you would expose as a COM object. After this code has been placed in a .apl file and added to your project, set the **Register for COM Interop** property to **True**. For more information, see How to: Register a Component for COM Interop.

Exposing Visual Visual APL objects to COM requires declaring a class interface, an events interface if it is required, and the class itself. Class members must follow these rules to be visible to COM:

- The class must be public.
- Properties, methods, and events must be public.
- Properties and methods must be declared on the class interface.
- Events must be declared in the event interface.

Other public members in the class that are not declared in these interfaces will not be visible to COM, but they will be visible to other .NET Framework objects.

To expose properties and methods to COM, you must declare them on the class interface and mark them with a **DispId** attribute, and implement them in the class. The order in which the members are declared in the interface is the order used for the COM vtable.

To expose events from your class, you must declare them on the events interface and mark them with a **DispId** attribute. The class should not implement this interface.

The class implements the class interface; it can implement more than one interface, but the first implementation will be the default class interface. Implement the methods and properties exposed to COM here. They must be marked public and must match the declarations in the class interface. Also, declare the events raised by the class here. They must be marked public and must match the declarations in the events interface.

Example

Visual APL

```
using System
using System.Runtime.InteropServices;
namespace project_name
{
    [ClassInterface(ClassInterfaceType.AutoDual)]
    [ProgId("TestCom.TestAdd")]
     public class TestCom {
        public TestCom() {
        }
        public function add(a, b) {
             return a+b
        }
     }
}
```

Namespaces (Visual APL Programming Guide)

Namespaces are heavily used in Visual APL programming in two ways. First, the .NET Framework uses namespaces to organize its many classes, as follows:

```
Visual APL

System.String.Format("Hello World!")
```

System is a namespace and **String** is a class contained within that namespace. The **using** keyword can be used so that the entire name is not required, like this:

```
Visual APL

using System;

String.Format("Hello");
```

For more information, see the topic using Directive (Visual APL Reference).

Second, declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the namespace keyword to declare a namespace, as in the following example:

```
Visual APL

namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

Namespaces Overview

A namespace has the following properties:

- They organize large code projects.
- They are delimited with the **.** operator.
- The **using directive** means you do not need to specify the name of the namespace for every class.

Using Namespaces (Visual APL Programming Guide)

Namespaces are heavily used within Visual APL programs in two ways. Firstly, the .NET Framework classes use namespaces to organize its many classes. Secondly, declaring your own namespaces can help control the scope of class and method names in larger programming projects.

Accessing Namespaces

Most Visual APL applications begin with a section of **using, refbyname** or **refbyfile** directives. This section lists the namespaces that the application will be using frequently, and saves the programmer from specifying a fully qualified name every time a method contained within is used.

For example, by including the line:

```
Visual APL

using System;


```

At the start of a program, the programmer can use the code:

```
Visual APL
String.Format("Hello, World!");
```

Instead of:

```
Visual APL
System.String.Format("Hello, World!");
```

Namespace Aliases

The using Directive (Visual APL Reference) can also be used to create an alias for a namespace. For example, if you are using a previously written namespace that contains nested namespaces, you might want to declare an alias to provide a shorthand way of referencing one in particular, like this:

```
Visual APL
using Co = Company.Proj.Nested;  // define an alias to represent a namespace
```

Using Namespaces to control scope

The **namespace** keyword is used to declare a scope. The ability to create scopes within your project helps organize code and provides a way to create globally-unique types. In the following example, a class entitled `SampleClass` is defined in two namespaces, one nested inside the other. The . Operator (Visual APL Reference) is used to differentiate which method gets called.

```
Visual APL
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            print "SampleMethod inside SampleClass";
        }
    }
    class SampleClassNext
```

```
    {
        public void SampleMethod()
        {
            print  "SampleMethod inside SampleClassNext");
        }
    }
    class Program
    {
        public function fn()
        {
            // Displays "SampleMethod inside SampleNamespace."
            outer = new SampleClass();
            outer.SampleMethod();

            // Displays "SampleMethod inside SampleNamespace."
            outer2 = new SampleClassNext();
            outer2.SampleMethod();
        }
    }
}
```

Fully Qualified Names

Namespaces and types have unique titles described by fully qualified names that indicate a logical hierarchy. For example, the statement `A.B` implies that `A` is the name of the namespace or type, and `B` is nested inside it.

In the following example, there are nested classes and namespaces. The fully qualified name is indicated as a comment following each entity.

```
Visual APL
namespace N1     // N1
{
    class C1      // N1.C1
    {
        class C2   // N1.C1.C2
        {
        }
    }
}
namespace N2  // N2
{
    class C2   // N2.C2
    {
    }

}
```

In the preceding code segment:

- The namespace `N1` is a member of the global namespace. Its fully qualified name is `N1`.
- The namespace `N2` is a member of global namespace. Its fully qualified name is `N2`.
- The class `C1` is a member of `N1`. Its fully qualified name is `N1.C1`.
- The class name `C2` is used twice in this code. However, the fully qualified names are unique. The first one is declared inside `C1`; thus, its fully qualified name is: `N1.C1.C2`.The second is declared inside a

namespace N2; thus, its fully qualified name is N2.C2.

Using the preceding code segment, you can add a new class member, C3, to the namespace N1.N2 as follows:

```
Visual APL
namespace N2
{
    class C3   // N2.C3
    {
    }
}
```

See the topic How to: Use the Namespace Alias Qualifier (Visual APL Programming Guide) for more details regarding the alias.

How to: Use the My Namespace (Visual APL Programming Guide)

The Microsoft.VisualBasic.MyServices namespace (**My** in Visual Basic) provides easy and intuitive access to a number of .NET Framework classes, enabling you to write code that interacts with the computer, application, settings, resources, and so on. Although originally designed for use with Visual Basic, the **MyServices** namespace can be used in Visual APL applications.

For more information about using the **MyServices** namespace from Visual Basic, see Development with My.

Adding a Reference

Before you can use the **MyServices** classes in your solution, you must add a reference to the Visual Basic library.

This is done using the keyword **refbyname** or **refbyfile**, in this example, we will use **refbyname**

This example calls various static methods contained in the **MyServices** namespace. For this code to compile, a reference to Microsoft.VisualBasic.DLL must be added to the project.

Visual APL

```
using System;
refbyname Microsoft.VisualBasic
using Microsoft.VisualBasic.Devices;
class TestMyServices
{
    public MyTest()
    {
        // Play a sound with the Audio class:
        myAudio = new Audio();
        print "Playing sound...";
        myAudio.Play(@"c:\WINDOWS\Media\chimes.wav");

        // Display time information with the Clock class:
        myClock = new Clock();
        print "Current day of the week: ";
        print myClock.LocalTime.DayOfWeek;
        print "Current date and time: ";
        print myClock.LocalTime;

        // Display machine information with the Computer class:
        myComputer = new Computer();
        print "Computer name: " + myComputer.Name;
        if (myComputer.Network.IsAvailable)
        {
            print "Computer is connected to network.";
        }
        else
        {
            print "Computer is not connected to network.";
        }
    }
}
```

Not all the classes in the **MyServices** namespace can be called from a Visual APL application: for example, the FileSystemProxy class is not compatible. In this particular case, the static methods that are part of FileSystem, which are also contained in VisualBasic.dll, can be used instead. For example, here is how to use

one such method to duplicate a directory:

```
Visual APL
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

Bottom of Form

Statements (Visual APL Programming Guide)

A statement is a procedural building-block from which all Visual APL programs are constructed. A statement can declare a local variable or constant, call a method, create an object, or assign a value to a variable, property, or field. A control statement can create a loop, such as a for loop, or make a decision and branch to a new block of code, such as an if or switch statement. Statements are usually terminated by a semicolon. For more information, see Statement Types (Visual APL Reference). The easiest way to think about statements, is that they do not return a value, whereas an expression will. The definition of a function is a statement, whereas invoking the function would be an expression.

A series of statements surrounded by curly braces form a block of code. A method body is one example of a code block. Code blocks often follow a control statement. Variables or constants declared within a function or method are only available to statements within the same function or method. Using the key word global variables created within a function can be exposed as global to the enclosing class. For example, the following code shows a method block and a code block following a control statement:

Visual APL
```
function IsPositive(number)
{
    if (number > 0) {
        return true;
    } else {
        return false;
    }
}
```

Statements in Visual APL often contain expressions. An expression in Visual APL is a fragment of code containing a literal value, a simple name, or an operator and its operands. Most common expressions, when evaluated, yield a literal value, a variable, or an object property or object indexer access. Whenever a variable, object property or object indexer access is identified from an expression, the value of that item is used as the value of the expression. In Visual APL, an expression can be placed anywhere that a value or object is required as long as the expression ultimately evaluates to the required type when strong typing is chosen.

Some expressions evaluate to a namespace, a type, a method group, or an event access. These special-purpose expressions are only valid at certain times, usually as part of a larger expression, and will result in a compiler error when used improperly.

Expressions (Visual APL Programming Guide)

An expression is a fragment of code that can be evaluated to a single value, object, method, or namespace. Expressions can contain a literal value, a method invocation, an operator and its operands, or a *simple name*. Simple names can be the name of a variable, type member, method parameter, namespace or type.

Expressions can use operators that in turn use other expressions as parameters, or method calls whose parameters are in turn other method calls, so expressions can range from simple to very complex.

Literals and Simple Names

The two simplest types of expressions are literals and simple names. A literal is a constant value that has no name. For example, in the following code example, both `5` and `"Hello World"` are literal values:

```
Visual APL
i = 5
s = "Hello World"
```

For more information on literals, see Types (Visual APL Reference).

In the example above, both `i` and `s` are simple names identifying local variables. When those variables are used in an expression, the value of the variable is retrieved and used for the expression. For example, in the following code example, when `DoWork` is called, the method receives the value 5 by default and is not able to access the variable `var`:

```
Visual APL
var = 5
DoWork(var)
```

Invocation Expressions

In the following code example, the call to `DoWork` is another kind of expression, called an invocation expression.

```
Visual APL
DoWork(var)
```

Specifically, calling a method is a method invocation expression. A method invocation requires the name of the method, either as a name as in the previous example, or as the result of another expression, followed by parenthesis and any method parameters. For more information, see Methods (Visual APL Programming Guide). A delegate invocation uses the name of a delegate and method parameters in parenthesis. For more information, see Delegates (Visual APL Programming Guide). Method invocations and delegate invocations evaluate to the return value of the method, if the method returns a value. Methods that return void will still return a null if the result of the method iis to be used in place of a value in an expression.

Remarks

Whenever a variable, object property, or object indexer access is identified from an expression, the value of that item is used as the value of the expression. An expression can be placed anywhere in Visual APL where a value or object is required, as long as the expression ultimately evaluates to the required type if strong typing is chosen, in the case of dynamic typing, the type is ambivalent.

Operators (Visual APL Programming Guide)

In Visual APL, an operator is a term or a symbol that takes one or more expressions, called operands, as input and returns a value. Operators that take one operand, such as the increment operator (**++**) or **new**, are called monadic operators. Operators that take two operands, such as arithmetic operators (**+**,**-**,*****,/) are called dyadic operators. One operator, the conditional operator (then else), takes three operands and is the sole tertiary operator in Visual APL.

The following Visual APL statement contains a single monadic operator, and a single operand. The increment operator, **++**, modifies the value of the operand `y`.:

Visual APL
```
y++;
```
The following Visual APL statement contains two dyadic operators, each with two operands. The assignment operator, **=**, has the integer `y`, and the expression `2 + 3` as operands. The expression `2 + 3` itself contains the addition operator, and uses the integer values `2` and `3` as operands:

Visual APL
```
y = 2 + 3;
```
An operand can be a valid expression of any size, composed of any number of other operations.

Operators in an expression are evaluated in a specific order, that is right to left. The following table divides the operators into categories based on the type of operation they perform.

Primary	x.y, f(x), a[x], x++, x--, new, typeof
Monadic (scalar and array)	+, -, !, ~, (T)x, ⍴, ×, ÷, ⍳, ∈, ⌊, ⌈, ↑, ↓
Dyadic (scalar and array)	(,ravel) , !, ?, ⍋, ⍒, ⌽, ⍉, ⊂, ⊃, ⍱, ⍲, ⍳
Arithmetic --- Multiplicative (scalar and array)	**×, ÷, |,** ⍟**, *,** ○
Arithmetic --- Multiplicative (scalar)	%
Arithmetic --- Additive (scalar and array)	+, -
Shift (scalar)	<<, >>
Relational (scalar and array)	<, >, <=, >=, ≤ ≥
Type testing (scalar)	is, as
Equality (scalar and array)	==, ≈ ≡ ≠ ≅, ≣
Equality (scalar)	!=
Logical (scalar and array)	⋀, ⋁, ⍲, ⍱, ~
Logical (scalar)	&, ^, |
Data Analysis (scalar and array)	⍳, ∈, ⌈, ⌊, ⊥, ⊤, !, ?, ⍋, ⍒, ⍷, ⌽, ⍉
Data Manipulation (scalar and array)	⍴, ↑ ↓ (,catenate), ⍪, ⊂, ⊃, ⍳, ⍱, ⍲
Operator Functions (scalar and array)	/, \, [], (. dot), ¨, ⍣, ⍤, ∘⍨
Conditional (Boolean)	&&, ||, then/else
Assignment	=, ← +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, …

Dyadic Operators are evaluated from right to left, Monadic (unary) operators are evaluated from left to right.

```
Visual APL
num1 = 5;
num1++;
print num1
```

However, the output of the following example code is undefined:

```
Visual APL
num2 = 5;
num2 = num2++;  //not recommended
print num2
```

Therefore, the latter example is not recommended. Parentheses can be used to surround an expression and force that expression to be evaluated before any others. For example, $2 \times 3 + 4$ would normally become 14. This is because dyadic operators evaluate from right to left . Writing the expression as $(2 \times 3) + 4$ results in 10, because it indicates to the Visual APL compiler that the multiplication operator $(\times)$ must be evaluated before the addition operator (**+**).

How to: Declare and Use Read/Write Properties (Visual APL Programming Guide)

Properties provide the convenience of public data members without the risks that come with unprotected, uncontrolled, and unverified access to an object's data. This is accomplished through *accessors*: special methods that assign and retrieve values from the underlying data member. The set accessor enables data members to be assigned, and the get accessor retrieves data member values.

This sample shows a `Person` class that has two properties: `Name` (string) and `Age` (int). Both properties provide **get** and **set** accessors, so they are considered read/write properties.

Example

Visual APL

```
class Person
{
    private m_name = "N/A";
    private m_Age = 0;

    // Declare a Name property of type string:
    public property Name {
        get {
            return m_name;
        }
        set {
            m_name = value;
        }
    }

    // Declare an Age property of type int:
    public property Age {
        get {
            return m_Age;
        }

        set {
            m_Age = value;
        }

    public override string ToString()
    {
        return "Name = " + this.Name + ", Age = " + this.Age;
    }
}

    public static function fn()
    {
        using System
        // Create a new Person object:
        person = new Person();

        // Print out the name and the age associated with the person:
        print String.Format("Person details - {0}", person);
```

```
        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        print String.Format("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        print String.Format("Person details - {0}", person);
    }
```

Output

```
Person details - Name = N/A, Age = 0
Person details - Name = Joe, Age = 99
Person details - Name = Joe, Age = 100
```

Robust Programming

In the previous example, the `Name` and `Age` properties are public and include both a **get** and a **set** accessor. This allows any object to read and write these properties. It is sometimes desirable, however, to exclude one of the accessors. Omitting the **set** accessor, for example, makes the property read-only:

```
Visual APL
public string Name
{
    get
    {
        return m_name;
    }
}
```

Once the properties are declared, they can be used as if they were fields of the class. This allows for a very natural syntax when both getting and setting the value of a property, as in the following statements:

```
Visual APL
person.Name = "Joe";
person.Age = 99;
```

Note that in a property **set** method a special `value` variable is available. This variable contains the value that the user specified, for example:

```
Visual APL
m_name = value;
```

Notice the clean syntax for incrementing the `Age` property on a `Person` object:

```
Visual APL
person.Age += 1;
```

If separate **set** and **get** methods were used to model properties, the equivalent code might look like this:

```
Cielo
person.SetAge(person.GetAge() + 1);
```

The **ToString** method is overridden in this example:

```
Visual APL
public override string ToString()
{
    return "Name = " + this.Name + ", Age = " + this.Age;
}
```

Notice that **ToString** is not explicitly used in the program. It is invoked by default by the **WriteLine** calls.

Exceptions and Exception Handling (Visual APL Programming Guide)

The Visual APL language's exception handling features provide a way to deal with any unexpected or exceptional situations that arise while a program is running. Exception handling uses the **try**, **catch**, and **finally** keywords to attempt actions that may not succeed, to handle failures, and to clean up resources afterwards. Exceptions can be generated by the common language runtime (CLR), by third-party libraries, or by the application code using the **throw** keyword.

In this example, a method tests for a division by zero, and catches the error. Without the exception handling, this program would terminate with a **DivideByZeroException was unhandled** error.

```
Visual APL
function SafeDivision(x, y)
{
    try
    {
        return (x ÷ y);
    }
    catch (DivideByZeroException dbz)
    {
        print "Division by zero attempted!";
        return 0;
    }
}
```

Exceptions Overview

Exceptions have the following properties:

- When your application encounters an exceptional circumstance, such as a division by zero or low memory warning, an exception is generated.
- Use a **try** block around the statements that might throw exceptions.
- Once an exception occurs within the **try** block, the flow of control immediately jumps to an associated exception handler, if one is present.
- If no exception handler for a given exception is present, the program stops executing with an error message.
- If a catch block defines an exception variable, you can use it to get more information on the type of exception that occurred.
- Actions that may result in an exception are executed with the **try** keyword.
- An exception handler is a block of code that is executed when an exception occurs. In Visual APL, the **catch** keyword is used to define an exception handler.
- Exceptions can be explicitly generated by a program using the **throw** keyword.
- Exception objects contain detailed information about the error, including the state of the call stack and a text description of the error.
- Code in a **finally** block is executed even if an exception is thrown, thus allowing a program to release resources.

Exception Handling (Visual APL Programming Guide)

A try block is used by Visual APL programmers to partition code that may be affected by an exception, and catch blocks are used to handle any resulting exceptions. A finally block can be used to execute code regardless of whether an exception is thrown -- which is sometimes necessary, as code following a try/catch construct will not be executed if an exception is thrown. A try block must be used with either a catch or a finally block, and can include multiple catch blocks. For example:

```
Visual APL
try
{
    // Code to try here.
}
catch (Exception ex)
{
    // Code to handle exception here.
}


Visual APL
try
{
    // Code to try here.
}
finally
{
    // Code to execute after try here.
}


Visual APL
try
{
    // Code to try here.
}
catch (Exception ex)
{
    // Code to handle exception here.
}
finally
{
    // Code to execute after try (and possibly catch) here.
}
```

A **try** statement without a **catch** or **finally** block will result in a compiler error.

Catch Blocks

A **catch** block can specify an exception type to catch. This type is called an *exception filter*, and must either be the Exception type, or derived from this type. Application-defined exceptions should derive from **ApplicationException**.

Multiple **catch** blocks with different exception filters can be chained together. Multiple **catch** blocks are evaluated from top to bottom, but only one **catch** block is executed for each exception thrown. The first **catch** block that species the exact type or a base class of the thrown exception will be executed. If no catch block specifies a matching exception filter, then a **catch** block with no filter (if any) will be executed. It is

important to place **catch** blocks with the most specific -- most derived -- exception classes first.

You should catch exceptions when:

- You have a specific understanding of why the exception was thrown, and can implement a specific recovery, such as catching a FileNotFoundException object and prompting the user to enter a new file name.
- You can create and throw a new, more specific exception. For example:

```
Visual APL
function GetInt(array, index)
{
    try
    {
        return array[index];
    }
    catch(IndexOutOfRangeException e)
    {
          throw ArgumentOutOfRangeException("Parameter index is out of
range.");
    }
}
```

- To partially handle an exception. For example, a **catch** block could be used to add an entry to an error log, but then re-throw the exception to allow subsequent handling to the exception. For example:

```
Visual APL
try
{
    // try to access a resource
}
catch (UnauthorizedAccessException e)
{
    LogError(e);  // call a custom error logging procedure
    throw e;      // re-throw the error
}
```

Finally Blocks

A **finally** block allows clean-up of actions performed in a **try** block. If present, the **finally** block executes after the **try** and **catch** blocks execute. A **finally** block is always executed, regardless of whether an exception is thrown or whether a **catch** block matching the exception type is found.

The **finally** block can be used to release resources such as file streams, database connections, and graphics handles without waiting for the garbage collector in the runtime to finalize the objects. See using Statement (Visual APL Reference) for more information.

In this example, the **finally** block is used to close a file opened in the **try** block. Notice that the state of the file handle is checked before it is closed. If the **try** block failed to open the file, the file handle will still be set to **null**. Alternatively, if the file is opened successfully and no exception is thrown, the **finally** block will still be executed and will close the open file.

```
Visual APL
fileinfo = new System.IO.FileInfo("C:\\file.txt");
try
{
  file = fileinfo.OpenWrite();
```

```
    file.WriteByte(0xF);
}
finally
{
    // check for null because OpenWrite
    // might have failed

    if (file != null)
    {
        file.Close();
    }
}
```

Using Exceptions (Visual APL Programming Guide)

In Visual APL, errors in the program at run time are propagated through the program using a mechanism called exceptions. Exceptions are thrown by code that encounters an error and caught by code that can correct the error. Exceptions can be thrown by the .NET Framework common language runtime (CLR) or by code in a program. Once an exception is thrown, it propagates up the call stack until a **catch** statement for the exception is found. Uncaught exceptions are handled by a generic exception handler provided by the system that displays a dialog box.

Exceptions are represented by classes derived from Exception. This class identifies the type of exception and contains properties with details about the exception. Throwing an exception involves creating an instance of an exception-derived class, optionally configuring properties of the exception, and then throwing the object with the **throw** keyword. For example:

```
Visual APL
function TestThrow()
{
    ex = ApplicationException("Demonstration exception in TestThrow()");
    throw ex;
}
```

After an exception is thrown, the runtime checks the current statement to see if it is within a **try** block. If so, any **catch** blocks associated with the **try** block are checked to see if they can catch the exception. **Catch** blocks normally specify exception types; if the type of the **catch** block is the same type as the exception, or a base class of the exception, the **catch** block can handle the method. For example:

```
Visual APL
function TestCatch()
{
    try
    {
        TestThrow();
    }
    catch (ApplicationException ex)
    {
        print ex.Message;
    }
}
```

If the statement that throws an exception is not within a **try** block or if the **try** block enclosing it has no matching **catch** block, the runtime checks the calling method for a **try** statement and **catch** blocks. The runtime continues up the calling stack, searching for a compatible **catch** block. After the **catch** block is found and executed, control is passed to the first statement after the **catch** block.

A **try** statement can contain more than one **catch** block. The first **catch** statement that can handle the exception is executed; any following **catch** statements, even if they are compatible, are ignored. It is also possible to also a part of the error message as a string as the catch condition. For example:

```
Visual APL
function TestCatch2()
{
    try
    {
        TestThrow();
    }
    catch (ApplicationException ex)
    {
```

```
        print ex.Message;  // this block will be executed
    }
    catch (Exception ex)
    {
        print ex.Message;  // this block will NOT be executed
    }
    print "Done";  // this statement is executed after the catch block
}
function TestCatch3() {
     try {
             ex = ApplicationException("my error")
             throw ex
     } catch ("my error") {
             print "my personal error"
     }
}
```

Before the **catch** block is executed, the **try** blocks that have been evaluated by the runtime, including the **try** block containing the compatible **catch** block, are checked for **finally** blocks. **Finally** blocks allow the programmer to clean up any ambiguous state that could be left over from an aborted **try** block, or to release any external resources (such as graphics handles, database connections or file streams) without waiting for the garbage collector in the runtime to finalize the objects. For example:

Visual APL

```
function TestFinally()
{
    try
    {
        a = ⍳3
        b = a[5]
    } catch (IndexOutOfRangeException ex) {
        print "Index Error"
    } finally {
        // reassign a to something bigger when exiting, even with index error
        print "resizing array in finally"
        a = ⍳10
    }
    try
    {
         b = a[5]
         print b
    }
    catch
    {
        print "error"
   }
}
```

In the first try block and index error is created, the catch display the Index Error message, then the finally runs, displaying the message and resizing the variable a. Then the second try block runs and the index is successful resulting in the value 5 being printed. The final catch is never run. The finally in the first try block is always run.

If no compatible **catch** block is found on the call stack after an exception is thrown, one of three things

happens:

- If the exception is within a destructor, the destructor is aborted and the base destructor, if any, is called.
- If the call stack contains a static constructor, or a static field initializer, a TypeInitializationException is thrown, with the original exception assigned to the InnerException property of the new exception.
- If the beginning of the thread is reached, the thread is terminated.

Creating and Throwing Exceptions (Visual APL Programming Guide)

Exceptions are used to indicate that an error has occurred while running the program. Exception objects that describe an error are created and then *thrown* with the throw keyword. The runtime then searches for the most compatible exception handler.

Programmers should throw exceptions when:

- The method cannot complete its defined functionality. For example, if a parameter to a method has an invalid value:

```
Visual APL
static void CopyObject(original)
{
    if (original == null)
    {
        throw new ArgumentException("Parameter cannot be null", "original");
    }
}
```

- An inappropriate call to an object is made, based on the object state. For example, trying to write to a read-only file. In cases where an object state does not allow an operation, throw an instance of InvalidOperationException or an object based on a derivation of this class. This is an example of a method that throws an **InvalidOperationException** object:

```
Visual APL
class ProgramLog
{
    logFile = null;
    public void OpenLog(System.IO.FileInfo fileName, System.IO.FileMode mode)
{}
    public void WriteLog()
    {
        if (!this.logFile.CanWrite)
        {
            throw new InvalidOperationException("Logfile cannot be
read-only");
        }
        // Else write data to the log and return.
    }
}
```

- When an argument to a method causes an exception. In this case, the original exception should be caught and an ArgumentException instance should be created. The original exception should be passed to the constructor of the **ArgumentException** as the InnerException parameter:

```
Visual APL
public static int GetValueFromArray(array, index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException ex)
```

```
    {
        argEx = new ArgumentException("Index is out of range", "index", ex);
        throw argEx;
    }
}
```

Exceptions contain a property called StackTrace -- this string contains the name of the methods on the current call stack, along with the file name and line number where the exception was thrown for each method. A **StackTrace** object is created automatically by the CLR from the point of the **throw** statement, so exceptions must be thrown from the point where the stack trace should begin.

All exceptions contain a property called Message -- this string should be set to explain the reason for the exception. Note that security sensitive information should not be put in the message text. In addition to **Message**, **ArgumentException** contains a property called ParamName that should be set to the name of the argument that caused the exception to be thrown. In the case of a property setter, **ParamName** should be set to `value`.

Public and protected methods should throw exceptions whenever they cannot complete their intended function. The exception class thrown should be the most specific exception available that fits the error conditions. These exceptions should be documented as part of the class functionality, and derived classes or updates to the original class should retain the same behavior for backwards compatibility.

Exceptions should not be used to alter the flow of a program as part of normal execution--they should only be used to report and handle error conditions. Exceptions should not be returned as a return value or parameter instead of being thrown. Programmers should not throw **System.Exception**, **System.SystemException**, **NullReferenceException** or **IndexOutOfRangeException** intentionally.

Defining Exception Classes

Programs can throw any of the predefined exception classes in the **System** namespace (except where previously noted), or create their own exception classes by deriving from ApplicationException. The derived classes should define at least four constructors -- one default constructor, one that sets the message property, and one that sets both the **Message** and **InnerException** properties. The fourth constructor is used to serialize the exception -- new exception classes should be serializable. For example:

```
Visual APL
public class InvalidDepartmentException : System.ApplicationException
{
   public InvalidDepartmentException() {}
   public InvalidDepartmentException(string message) {}
   public InvalidDepartmentException(string message, System.Exception inner) {}

   // Constructor needed for serialization
   // when exception propagates from a remoting server to the client.
   protected InvalidDepartmentException(System.Runtime.Serialization.SerializationInfo info,
      System.Runtime.Serialization.StreamingContext context) {}
}
```

New properties should only be added to the exception class when the data they provide is useful to resolving the exception. If new properties are added to the derived exception class, `ToString()` should be overridden to return the added information.

Creating Custom Attributes (Visual APL Programming Guide)

You can create your own custom attributes by defining an attribute class, a class that derives directly or indirectly from Attribute, which makes identifying attribute definitions in metadata fast and easy. Suppose you want to tag classes and structs with the name of the programmer who wrote the class or struct. You might define a custom `Author` attribute class:

```
Visual APL
[AttributeUsage(System.AttributeTargets.Class)]
public class Author : System.Attribute
{
    private name;
    public double version;

    public Author(string name)
    {
        this.name = name;
        this.version = 1.0;
    }
}
```

The class name is the attribute's name, `Author`. It is derived from **System.Attribute**, so it is a custom attribute class. The constructor's parameters are the custom attribute's positional parameters, in this case, `name`, and any public read-write fields or properties are named parameters, in this case, `version` is the only named parameter. Note the use of the **AttributeUsage** attribute to make the `Author` attribute valid only on **class** and **struct** declarations.

You could use this new attribute as follows:

```
Visual APL
[Author("H. Ackerman", version = 1.1)]
class SampleClass
{
    // H. Ackerman's code goes here...
}
```

AttributeUsage has a named parameter, **AllowMultiple**, with which you can make a custom attribute single-use or multiuse.

```
Visual APL
[AttributeUsage(System.AttributeTargets.Class | AllowMultiple = true)  //
multiuse attribute
]
public class Author : System.Attribute


Visual APL
[Author("H. Ackerman", version = 1.1)]
[Author("M. Knott", version = 1.2)]
class SampleClass
{
    // H. Ackerman's code goes here...
    // M. Knott's code goes here...
}
```

Compiler-Generated Exceptions (Visual APL Programming Guide)

Some exceptions are thrown automatically by the .NET Framework's common language runtime (CLR) as a result of basic operations that fail. These exceptions and their error conditions are listed below.

Exception	Description
ArithmeticException	A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException.
ArrayTypeMismatchException	Thrown when an array cannot store a given element because the actual type of the element is incompatible with the actual type of the array.
DivideByZeroException	Thrown when an attempt is made to divide an integral value by zero.
IndexOutOfRangeException	Thrown when an attempt is made to index an array when the index is less than zero or outside the bounds of the array.
InvalidCastException	Thrown when an explicit conversion from a base type to an interface or to a derived type fails at runtime.
NullReferenceException	Thrown when you attempt to reference an object whose value is null.
OutOfMemoryException	Thrown when an attempt to allocate memory using the new operator fails. This indicates that the memory available to the Common Language Runtime has been exhausted.
OverflowException	Thrown when an arithmetic operation in a `checked` context overflows.
StackOverflowException	Thrown when the execution stack is exhausted by having too many pending method calls; usually indicates a very deep or infinite recursion.
TypeInitializationException	Thrown when a static constructor throws an exception and no compatible `catch` clause exists to catch it.

How to: Handle an Exception Using try/catch (Visual APL Programming Guide)

The purpose of a try-catch block is to catch and handle an exception generated by working code. Some exceptions can be handled in a **catch** block and the problem solved without the exception being re-thrown; however, more often the only thing you can do is make sure the appropriate exception is thrown.

In this example, IndexOutOfRangeException is not the most appropriate exception: ArgumentOutOfRangeException makes more sense for the method because the error is caused by the `index` argument passed in by the caller.

```
Visual APL
function GetInt(array, index)
    {
        try
        {
            return array[index];
        }
        catch (IndexOutOfRangeException e)  // CS0168
        {
            print e.Message;
            //set IndexOutOfRangeException to the new exception's
InnerException
            throw new ArgumentOutOfRangeException("index parameter is out of
range.", e);
        }
    }
```

Comments

The code that results in an exception is enclosed in the **try** block. A **catch** statement is added immediately after to handle `IndexOutOfRangeException`, if it occurs. The **catch** block handles the `IndexOutOfRangeException` and throws the more appropriate `ArgumentOutOfRangeException` exception instead. In order to provide the caller with as much information as possible, consider specifying the original exception as the InnerException of the new exception. Because the **InnerException** property is readonly, you must assign it in the constructor of the new exception.

Using Strings (Visual APL Programming Guide)

A Visual APL string is an array of characters declared using the **string** keyword, and is the native .Net string object. A string literal is declared using quotation marks, as shown in the following example:

```
Visual APL
s = "Hello, World!"
```

You can extract substrings, and concatenate strings, like this:

```
Visual APL
s1 = "orange";
s2 = "red";

s1 += s2;
print s1  // outputs "orangered"

s1 = s1.Substring(2, 5);
print s1  // outputs "anger"
```

String objects are *immutable*, meaning that they cannot be changed once they have been created. Methods that act on strings actually return new string objects. In the previous example, when the contents of `s1` and `s2` are concatenated to form a single string, the two strings containing `"orange"` and `"red"` are both unmodified. The **+=** operator creates a new string that contains the combined contents. The result is that `s1` now refers to a different string altogether. A string containing just `"orange"` still exists, but is no longer referenced when `s1` is concatenated.

Note

Use caution when creating references to strings. If you create a reference to a string, and then "modify" the string, the reference will continue to point to the original object, not the new object that was created when the string was modified. The following code illustrates the danger:

```
Visual APL
s1 = "Hello";
s2 = s1;
s1 += " and goodbye.";
print s2 //outputs "Hello"
```

Because modifications to strings involve the creation of new string objects, for performance reasons, large amounts of concatenation or other involved string manipulation should be performed with the StringBuilder class, like this:

```
Visual APL
System.Text.StringBuilder sb = new System.Text.StringBuilder();
sb.Append("one ");
sb.Append("two ");
sb.Append("three");
str = sb.ToString();
```

The **StringBuilder** class is discussed further in the "Using Stringbuilder" section.

Working with Strings

Escape Characters

Escape characters such as "\n" (new line) and "\t" (tab) can be included in strings. The line:

```
Visual APL
hello = "Hello\nWorld!";
```

is the same as:

```
Hello

World!
```

If you want to include a backward slash, it must be preceded with another backward slash. The following string:

```
Visual APL
filePath = "\\\\My Documents\\";
```

is actually the same as:

```
\\My Documents\
```

The @ Symbol

The **@** symbol tells the string constructor to ignore escape characters and line breaks. The following two strings are therefore identical:

```
Visual APL
p1 = "\\\\My Documents\\My Files\\";
p2 = @"\\My Documents\My Files\";
```

ToString()

Like all objects derived from Object, strings provide the ToString method, which converts a value to a string. This method can be used to convert numeric values into strings, like this:

```
Visual APL
year = 1999;
msg = "Eve was born in " + year.ToString();
print msg  // outputs "Eve was born in 1999"
```

Accessing Individual Characters

Individual characters contained in a string can be accessed using methods such as **SubString()**, **Replace(), Split()** and **Trim()**.

```
Visual APL
s3 = "Visual Visual APL Is Cool";

print s3.Substring(7, 5))         // outputs "Visual APL"
print s3.Replace("Cool", "Marvelous"))  // outputs "Visual Basic Express"
```

It is also possible to copy the characters into a character array, like this:

```
Visual APL
s4 = "Hello, World";
arr = s4.ToCharArray(0, s4.Length);

foreach (c in arr)
{
    print c  // outputs "Hello, World"
}
```

Individual characters from a string can be accessed with an index, like this:

```
Visual APL
s5 = "Printing backwards";

for (i = 0; i < s5.Length; i++)
```

```
{
    print s5[(s5.Length - i) - 1];  // outputs "sdrawkcab gnitnirP"
}
```

Changing Case

To change the letters in a string to upper or lower case, use **ToUpper()** or **ToLower()**, like this:

```
Visual APL
string s6 = "Battle of Hastings, 1066";

print s6.ToUpper()  // outputs "BATTLE OF HASTINGS 1066"
print s6.ToLower()  // outputs "battle of hastings 1066"
```

Comparisons

The simplest way to compare two strings is to use the **==** and **!=** operators, which perform a case-sensitive comparison.

```
Visual APL
color1 = "red";
color2 = "green";
color3 = "red";

if (color1 == color3)
{
    print "Equal";
}
if (color1 != color2)
{
    print "Not equal";
}
```

String objects also have a **CompareTo()** method that returns an integer value based on whether one string is less-than (<)or greater-than (>) another. When comparing strings, the Unicode value is used, and lower case has a smaller value than upper case.

```
Visual APL
s7 = "ABC";
s8 = "abc";

if (s7.CompareTo(s8) > 0)
{
    print "Greater-than";
}
else
{
    print "Less-than";
}
```

To search for a string inside another string, use **IndexOf()**. **IndexOf()** returns -1 if the search string is not found; otherwise, it returns the zero-based index of the first location at which it occurs.

```
Visual APL
s9 = "Battle of Hastings, 1066";

print s9.IndexOf("Hastings");  // outputs 10
print s9.IndexOf("1967");      // outputs -1
```

Splitting a String into Substrings

Splitting a string into substringsâ€”such as splitting a sentence into individual wordsâ€”is a common programming task. The **Split()** method takes a **char** array of delimiters, for example, a space character, and returns an array of substrings. You can access this array with **foreach**, like this:

```
Visual APL
delimit = ' ' ;
string s10 = "The cat sat on the mat.";
foreach (substr in s10.Split(delimit))
{
    print substr;
}
```

This code outputs each word on a separate line, like this:

```
The

cat

sat

on

the

mat.
```

Null Strings and Empty Strings

An empty string is an instance of a System.String object that contains zero characters. Empty strings are used quite commonly in various programming scenarios to represent a blank text field. You can call methods on empty strings because they are valid **System.String** objects. Empty strings are initialized like this:

```
string s = "";
```

By contrast, a null string does not refer to an instance of a **System.String** object and any attempt to call a method on a null string results in a NullReferenceException. However, you can use null strings in concatenation and comparison operations with other strings. The following examples illustrate some cases in which a reference to a null string does and does not cause an exception to be thrown:

```
str = "hello";
nullStr = null;
emptyStr = "";

tempStr  = str + nullStr; // tempStr = "hello"
b = (emptyStr == nullStr);// b = false;
emptyStr + nullStr = ""; // creates a new empty string
i  = nullStr.Length; // throws NullReferenceException
```

Using StringBuilder

The **StringBuilder** class creates a string buffer that offers better performance if your program performs a lot of string manipulation. The **StringBuilder** string also allows you to reassign individual characters, something the built-in string data type does not support. This code, for example, changes the content of a string without creating a new string:

```
Visual APL
sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
```

```
print sb.ToString();
```

In this example, a **StringBuilder** object is used to create a string from a set of numeric types:

```
Visual APL
class TestStringBuilder
{
    public static void test()
    {
        using System
        sb = new System.Text.StringBuilder();

        // Create a string composed of numbers 0 - 9
        for (i = 0; i < 10; i++)
        {
            sb.Append(i.ToString());
        }
        print sb;  // displays 0123456789

        // Copy one character of the string (not possible with a
System.String)
        sb[0] = sb[9];

        print sb;  // displays 9123456789
    }
}
```

How to: Parse Strings Using the Split Method (Visual APL Programming Guide)

The following code example demonstrates how a string can be parsed using the System.String.Split method. This method works by returning an array of strings, where each element is a word. As input, Split takes an array of chars that indicate which characters are to be used as delimiters. In this example, spaces, commas, periods, colons, and tabs are used. An array containing these delimiters is passed to **Split**, and each word in the sentence is displayed separately using the resulting array of strings.

Example

Visual APL

```
function fn() {
        delimiterChars = ' ' ',' '.' ':' '\t'
        text = "one\ttwo three:four,five six seven"
        print String.Format("Original text: '{0}'", text)
        words = text.Split(delimiterChars, text.Length)
        print String.Format("{0} words in text:", words.Length)
        foreach (s in words) {
               print s
        }
}
```

Output

Visual APL

```
Original text: 'one two three:four,five six seven' 7 words in text: one two
three four five six seven
```

How to: Join Multiple Strings (Visual APL Programming Guide)

There are two ways to join multiple strings: using the **+** operator that the String class overloads, and using the StringBuilder class. The **+** operator is easy to use and makes for intuitive code, but it works in series; a new string is created for each use of the operator, so chaining multiple operators together is inefficient. For example:

```
Visual APL
two = "two";
str = "one " + two + " three";
```

Although four strings appear in the code, the three strings being joined and the final string containing all three, five strings are created in total because the first two strings are joined first, creating a string containing "one two." The third is appended separately, forming the final string stored in `str`.

Alternatively, the **StringBuilder** class can be used to add each string to an object that then creates the final string in one step. This strategy is demonstrated in the following example.

Example

The following code uses the Append method of the **StringBuilder** class to join three strings without the chaining effect of the **+** operator.

```
Visual APL
    function fn()
    {
        two = "two";

        sb = new System.Text.StringBuilder();
        sb.Append("one ");
        sb.Append(two);
        sb.Append(" three");
        print sb.ToString();

        str = sb.ToString();
        print str;
    }
```

How to: Search Strings Using Regular Expressions (Visual APL Programming Guide)

The System.Text.RegularExpressions.Regex class can be used to search strings. These searches can range in complexity from very simple to making full use of regular expressions. The following are two examples of string searching using the **Regex** class. For more information, see .NET Framework Regular Expressions.

Example

The following code is a console application that performs a simple case insensitive search of the strings in an array. The static method System.Text.RegularExpressions.Regex.IsMatch(System.String,System.String,System.Text.RegularExpressions.RegexOptions) performs the search given the string to search and a string containing the search pattern. In this case, a third argument is used to indicate that case should be ignored. For more information, see System.Text.RegularExpressions.RegexOptions.

Visual APL

```
    function fn()
    {
        sentences = (
            "cow over the moon"
            "Betsy the Cow"
            "cowering in the corner"
            "no match here"
        );

        sPattern = "cow";

        foreach (s in sentences)
        {
            print String.Format("{0,24}", s);

            if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
            {
                print String.Format("  (match for '{0}' found)", sPattern);
            }
            else
            {
                print String.Format();
            }
        }
    }
```

Output

```
       cow over the moon  (match for 'cow' found)
           Betsy the Cow  (match for 'cow' found)
           Betsy the Cow  (match for 'cow' found)
  cowering in the corner  (match for 'cow' found)
           no match here
```

The following code is a console application that uses regular expressions to validate the format of each string in an array. The validation requires that each string take the form of a telephone number in which three

groups of digits are separated by dashes, the first two groups contain three digits, and the third group contains four digits. This is accomplished with the regular expression ^\\d{3}-\\d{3}-\\d{4}$. For more information, see Regular Expression Language Elements.

```
Visual APL
    function fn()
    {
        numbers = (
            "123-456-7890"
            "444-234-22450"
            "690-203-6578"
        "146-893-232"
            "146-839-2322"
            "4007-295-1111"
            "407-295-1111"
            "407-2-5555"
        );

        string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";

        foreach (s in numbers)
        {
            res = String.Format("{0,14}", s);
            if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
            {
                print res+String.Format(" - valid");
            }
            else
            {
                print res+String.Format(" - invalid");
            }
        }
    }
```

Output

```
    123-456-7890 - valid
    444-234-22450 - invalid
    690-203-6578 - valid
    146-893-232 - invalid
    146-839-2322 - valid
    4007-295-1111 - invalid
    407-295-1111 - valid
    407-2-5555 - invalid
```

How to: Search Strings Using String Methods (Visual APL Programming Guide)

The string type, which is an alias for the System.String class, provides a number of useful methods for searching the contents of a string. The following example uses the IndexOf, LastIndexOf, StartsWith, and EndsWith methods.

Example

Visual APL

```
    function fn()
    {
        str = "A silly sentence used for silly purposes.";
        print String.Format("'{0}'",str);

        test1 = str.StartsWith("a silly");
        print String.Format("starts with 'a silly'? {0}", test1);

        test2 = str.StartsWith("a silly",
System.StringComparison.OrdinalIgnoreCase);
        print String.Format("starts with 'a silly'? {0} (ignoring case)",
test2);

        test3 = str.EndsWith(".");
        print String.Format("ends with '.'? {0}", test3);

        first = str.IndexOf("silly");
        last = str.LastIndexOf("silly");
        str2 = str.Substring(first, last - first);
        print String.Format("between two 'silly' words: '{0}'", str2);
    }
```

Output

```
'A silly sentence used for silly purposes.'
starts with 'a silly'? False
starts with 'a silly'? True (ignore case)
ends with '.'? True
between two 'silly' words: 'silly sentence used for '
```

Collection Classes (Visual APL Programming Guide)

The .NET Framework provides specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces, and these interfaces may be inherited to create new collection classes that fit more specialized data storage needs.

```
Visual APL
list = ArrayList();
list.Add(10);
list.Add(20);
```

Collection Classes Overview

Collection Classes have the following properties

- Collection classes are defined as part of the **System.Collections** or **System.Collections.Generic** namespace.
- Most collection classes derive from the interfaces **ICollection**, **IComparer**, **IEnumerable**, **IList**, **IDictionary**, and **IDictionaryEnumerator** and their generic equivalents.

Native File Access

Native file access is provided through a set of system functions. These provide the ability to create, open, close, remove, resize, and add and retrieve serializable data to/from native files on the machines hard disk.

The File, Path, Directory, etc classes available in the System.IO assembly provides similar functionality, but with far greater detailed control.

To use the Native File System in your application, you will need to add a reference to the Visual APL Share/Native File System assembly. Here is an example of "referencing" and "using" the assembly by its strong name:

```
refbyname APLNext.APL.LegacyOps
using APLNext.Legacy.NativeFileSystem
```

□nappend

Appends data to a native file which is associated with the tie number. Any serializable object can be appended to a file using this system function.

```
      how are you  □nappend ¯1          "
```

ncreate

Creates a native file in the specified directory or in the current directory, if no directory is given. Tie numbers for referencing native files are negative to avoid a conflict with the positive tie numbers used by the component file system.

```
      c: \\mytest\\test.nf  n"create -1          "
-1
      test1.nf  ncreate -2 "        "
-2
     nnums
-1 -2
     nnames
 c: \\mytest\\test.nf   c: \\mydefault\\test1.nf"                    "
```

Note

The double \ as the \ is used as a delimiter in strings. To avoid having to double the \ you can use @ at the beginning of a string.

```
a = @ c: \mytest\test.nf      "                  "
```

Which will place the raw string in the variable "a".

You can also create a file and let the system assign the tie number.

For instance:

```
     ncreate  c: \\mytest\\test2.nf  "                    "
-3
```

The system will also assign the next available tie number if you specify a tie number of 0.

For instance:

```
      c: \\mytest\\test.nf  n"create 0           "
-3
```

□nerase

This will delete the specified native file permanently.

```
c: \\mytest\\test2.nf  □nerase ¯3          "
```

```
□nerase  c: \\mytest\\test2.nf  "                    "
```

nnames

Returns a string array of native file names which are currently tied.

nnums

Returns an integer array of tie numbers associated with native files which are tied.

□nread

Reads data from a native file which is tied and associated with a given tie number.

```
a = □nread tn, convert, numberOfBytes, beginOffSet
```

The standard conversion values are:

Code	Description
11:	boolean (true/false, not bit)
81:	bytes
82:	chars (compatible with 82 in existing system)
83:	string (compatible with 82 in existing system)
163:	short (Int16, 16 bit integer)
164:	ushort (UInt16, unsigned short)
323:	int (Int32, 32 bit integer, default)
324:	uint (UInt32, unsigned int)
325:	float (Single, 32 bit real)
643:	long (Int64, 64 bit integer)
644:	ulong (UInt64, unsigned long)
645:	double (Double, 64 bit real, default)
1285:	Decimal (128 bit real)
807:	object (serialized object)

Example:

```
a = □nread ¯2 82 10 0
```

Reads back 10 characters.

convert can also be a TypeCode:

```
a = □nread ¯2 TypeCode.Char 10 0
```

convert can also be an intrinsic Type

```
a = □nread ¯2 Char 10 0
```

For reading matrices and arrays of heterogeneous data or any serialized object, use 807:

```
a = □nread ¯2 807 10 0
```

Note

□nsize tn returns a long, which may not be supported in your operator set. The default operator set does not include the long type. This can cause an error when catenating. So use spaces instead of commas, or cast the result to integer, as example:

```
a = □nread ¯2 82 (□nsize ¯2) 0
or
a = □nread ¯2,82,((int) □nsize ¯2), 0
```

nrename

Rename a native file currently tied and associated with a tie number.

```
new_filename nrename tn
```

Where new_filename is the new filename and tn is the existing tie number.

⎕nreplace

Replace existing data in a native file, beginning at the location given and continuing until all of the provided data is written.

```
      100.1 ⎕nreplace ¯3,10
      10L 10 ⎕nreplace ¯3,10
      10.1  test  10 ⎕nreplace ¯3,1⍕0    "
      (3 3⍴⍳9) ⎕nreplace ¯3,10
```

Note that a nested array or matrix is serialized and written to the file. To read serialized data back from the file use the 807 code.

□nresize

Resizes the native file associated with the tie number to a new size in bytes. The new size can be 0, smaller, larger or the same size as the existing size.

```
      0 □nresize ¯2
      10000 □nresize ¯2
```

The resize does not change the data, but making the file smaller will result in the loss of data that existed beyond the new file size.

nulls are used to pad the file when a resize makes the file larger □a v[□i o]

□nsize

Returns a long which represents the size of the file.

```
A = □nsize ¯3
```

If you want the size to be an Int32 use:

```
A = (int)□nsize ¯3
```

nuntie

Unties the native file associated with the tie number.

```
nuntie -3
```

ntie

Ties a native file in the specified directory or in the current directory, if no directory is given. Tie numbers for referencing native files are negative to avoid a conflict with the positive tie numbers used by the component file system.

```
      c: \\mytest\\test.nf  ntie -1              "
-1
      test1.nf  ntie -2    "        "
-2
     nnums
-1 -2
     nnames
 c: \\mytest\\test.nf   c: \\mydefault\\test1.nf"                    "
```

Note

The double \ as the \ is used as a delimiter in strings. To avoid having to double the \ you can use @ at the beginning of a string.

```
a = @ c: \mytest\test.nf     "                  "
```

Which will place the raw string in the variable "a".

You can also tie a file and let the system assign the tie number.

For instance:

```
     ntie  c: \\mytest\\test2.nf  "                    "
-3
```

The system will also assign the next available tie number if you specify a tie number of 0.

For instance:

```
      c: \\mytest\\test.nf  ntie 0               "
-3
```

Access Attributes

You can specify what permissions to tie the target file with by supplying a second element to the right argument.

This second argument is the sum of the permissions to request and allow for the file tie operation.

If the access attributes element is not specified, then the default value is 2 (Read/Write access, Exclusive tie)

Here is the list of the valid tie permission request values. The sum of the requested access attributes number can contain only one of these values:

Code	Description
0:	Request read access
1:	Request write access
2:	Request read and write access.

Here is the list of values which control what permissions are granted to future tie requests for the file being tied. The sum of the requested access attributes number can contain only one of these values:

Code	Description
0:	Compatibility Mode
16:	Exclusive Tie, no other ties can be made to the file.
32:	Read access is granted to future ties.
48:	Write access is granted to future ties.
64:	Read and Write access is granted to future ties.

Here is an example of tieing files with different Request and Granted permissions:

```
// tie with read access, and compatibility mode
 c: \\mytest\\test.nf  "tie -1 0           "

// tie with read/write access, and grant read/write
 c: \\mytest\\test1.nf  "tie -1 (2+64)      "

// tie with read/write access, and grant no permissions
 c: \\mytest\\test2.nf  "tie -1 (2+16)      "
```

ncopy

Copy the contents of the specified source file to the specified target path.

```
ncopy source_filepath target_filepath
```

Where source_filename is source file from which to copy data, and target_filename is the target path of the copy operation.

nexists

Returns a value of 1 or 0 indicating if the specified file name exists. Specifying a directory name without a file returns 0 (false).

```
      A = @'c:\Windows\0.log'
      nexists A
1
      B = @'c:\Windows\'
      nexists B
0
```

□nstream

Returns the underlying .Net FileStream object for the associated tie number. This allows the use of all features provided by the FileStream object, while still maintaining compatibility with the Native File system.

```
      fs = □nstream ¯3

      fs.CanRead
true
      fs.CanWrite
true
```

Session Commands (Visual APL)

The Cielo Explorer includes a wide range of commands for managing the various aspects of the session.

These aspects include the listing of session contents, script management, and editing of variables.

)cd

Changes the current context of the session into or out of the specified object.

Syntax:

```
)cd obj
```

obj: The name of a class, variable, or path control string.

Remarks:

This session command provides the ability to explore classes. It is possible to explore either an instance of a class or the class itself.

When this session command is used without an argument it displays the current class being explored, for instance when at the top level, in the session, this is displayed

```
      )cd
session
```

The right argument to the)cd session command is either a classname, a variable or a relative path.

To navigate to a particular class:

```
      )cd classname
```

To navigate to an instance of a class:

```
      a = classname()
      )cd a
```

To navigate to a relative location:

```
      )cd ../../a/b/c
```

Or to navigate up one level:

```
      )cd ..
```

To return to the session or root:

```
```

```
      )cd /
```

Examples and narrative:

Once you have navigated into a class, you will see all of the methods, properties, events and fields in the class, regardless of member attributes. This means you can review members which are public, internal, private, etc.

As an example, consider an integer:

```
      a = 10
      )cd a
Loaded instance of:  System.Int32
```

Now if we look at the)fns in this instance of the ValueType Int32 we see:

```
      )fns
_dataRepresentation CompareTo           Equals              Finalize
          GetHashCode
GetType             GetTypeCode         MemberwiseClone     Parse
              ToString
TryParse
```

The)fns includes methods, functions and the methods associated with properties.

Now if we look at)vars we see:

```
     )vars

m_value  MaxValue MinValue
```

We can navigate back up to the session by entering)cd ..

```
     )cd ..
Loaded instance of:  APLNext.APL.Objects.module
```

If we try to navigate up again:

```
     )cd ..
Current instance is session
```

We see that we are already at the session level, and cannot navigate further up.

While we are back at the session level, let’s consider what is visible on the Int32 we have placed in the

variable a.

If we look at intellisense on an instance of an Int32, we see a small subset of those members we saw when we navigated into the instance of Int32 on the variable a.

Specifically, if we navigate back into the a variable:

```
     )cd a
Loaded instance of:  System.Int32
```

If we then check the variable m_value, which is not normally available to investigate, we find:

```
     m_value
10
```

So we see that the Int32 is an object, a ValueType in particular, and that the integer value is stored on the field m_value.

If we want to know where we are in our navigation, we can always do)cd without an argument:

```
     )cd
session/a
```

No matter how deep we have navigated we can always move back to the session by entering:

```
    )cd /
Current instance is session
```

To review, everything is an object, and we can navigate through those objects using)cd, in this example lets look at an Int32 and navigate down and up through this object.

```
     a = 10
     )cd a
Loaded instance of:  System.Int32

      )vars
m_value  MaxValue MinValue

      m_value
10

      )cd MaxValue
Loaded instance of:  System.Int32

      )vars
m_value  MaxValue MinValue

      m_value
```

```
2147483647

      )cd
session/a/MaxValue

      )cd ..
Loaded instance of:  System.Int32

      )cd
session/a

      )cd ..
Loaded instance of:  APLNext.APL.Objects.module

      )cd
session
```

)classes

Shows the current list of classes which have been defined in the session.

Syntax:

```
)classes
```

Remarks:

The *)classes* command shows the list of classes which have been created in the session.

Classes are most commonly created in the session by running a script file.

Example:

Here is an example script which contains the definition of two classes:

```
// Script: sc1

public class math {
    function add(a, b) {
        return a + b
    }

    function subtract(a, b) {
        return a - b
    }
}

public class useMath {
    function fn(a, b) {
        m = math()
        return m.add(a, b)
    }
}
```

Now lets run the script to create the classes in the session:

```
     // display the contents of the )classes list
     )classes

     // the list is empty

     // load and run the script 'sc1'
     )load sc1

     // display )classes again
     )classes
math usemath
     // the two classes now exist in the session.

     // run the useMath class:
     um = useMath()
     um.fn(10, 20)
30
```

)clear

Clears all variables, functions, etc, from the active Cielo Explorer.

Syntax:

```
)clear
```

Remarks:

When the clear session command is run in the Cielo Explorer, the .Net AppDomain which currently represents the Cielo Explorer is shutdown, thus removing from memory any variables, functions, UDF's, file ties, etc, from memory.

If an assembly is referenced into the session by the use of the refbyname or refbyfile directives, then the DLL which that reference represents is tied in memory by the session. This behavior is required by the .Net security model.

When the Cielo Explorer AppDomain is unloaded by the clear command, any assembly references are also removed from memory, meaning that any tied assemblies can again be modified, recreated, and moved on the disc.

Example:

```
     a = 10 20 30
     a
10 20 30

     )clear

     a
The variable 'a' does not exist
```

)edit

Opens a script file for editing in the current Session Project of the active Solution.

Syntax:

```
)edit script
```

script: The name of the script file to create or open.

Remarks:

When the edit command is run in the Cielo Explorer session, a script file of the specified name is opened for editing from the current Session Project in Visual Studio, and given the active window focus.

If the script file does not exist in the current Session Project, then a new script file is created in the Session Project by the supplied name, and opened for editing.

If the script file does exist in the current Session Project, then it is opened and focused for editing.

If the script file is already open in Visual Studio, then that script file is brought to the forefront and receives the window focus.

Saving a script

Once you have edited the contents of a script file, you can save and execute the script to the Cielo Explorer session by pressing the key sequence **Ctrl+E+E**. Once the script is saved and executed to the Cielo Explorer, a message is printed to the session stating that the script was modified and imported.

Example:

```
     // make a variable in the session
     a = 10 20 30
     a
10 20 30

     // edit a new script
     )edit sc

     // add this line to the script
     a = a + 100

     // save the script by pressing Ctrl+E+E.

     // this line is printed to the session
     Script 'sc' updated

     // now again display the contents of 'a' in the session.
     a
110 120 130
```

)fns

Shows the current list of functions which have been defined in the session.

Syntax:

```
)fns
```

Remarks:

The *)fns* command shows the list of functions which have been created in the session.

Here are a few examples of creating functions in the session:

- Entering its declaration directly in the session:

```
     // display the contents of the )fns list
     )fns

     // the list is empty

     // define a new function called 'add'
     function add(a, b) { return a + b }

     // display )fns again
     )fns
add
     // the function 'add' is now in the list.
```

- Using □def to declare a function from a text string:

```
     // display the contents of the )fns list
     )fns

     // the list is empty

     // define a function from a string
     □def  function add(a, b) { return a + b }                    "
true

     // display )fns again
     )fns
add
     // the function 'add' is now in the list.
```

- Execute a script which contains the definition of one or more functions. Here is an example script which contains the definition of two functions:

```
// Script: sc1

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

```
function subtract(a, b) {
      return a - b
}
```

Now lets run the script to create the functions in the session:

```
     // display the contents of the )fns list
     )fns

     // the list is empty

     // load and run the script 'sc1'
     )load sc1

     // display )fns again
     )fns
add subtract
     // the functions are now in the list.
```

These are only a few simple examples of creating functions in the session. Any valid expression or statement which creates a function can be run is the session, and once that command is run the resultant function will be present in the *)fns* list.

Created Function Time Stamping

When functions are dynamically created in the session, they receive an associated DateTime object which represents the moment that the function was created in the session. There are several system quad functions which allow the retrieval and modification of this time stamp.

Example:

```
     // check the contents of the )fns list
     )fns
mult sub
     // there are two functions currently defined

     // define a new function called 'add'
     function add(a, b) { return a + b }

     // check the )fns list
     )fns
add mult sub
     // the function 'add' is now in the list.

     // try running add:
     10 add 20
30
```

)load

Loads and executes a script from the current Session Project.

Syntax:

```
)load script
```

script: The name of the script file to load.

Remarks:

The)load command looks in the current Session Project for a script named the specified name. If the script file exists, it is added to the)scripts list in the session, and then the contents of the script are executed in the session.

This command has the same behavior as pressing **Ctrl+E+E** in an open script in Visual Studio.

Example:

```
     // check if 'a' exists
     a
name 'a' is not defined

     // check the contents of the )script list
     )scripts

     // the )scripts list is empty

     // load a script which defines 'a'
     )load sc

     // check if 'a' exists
     a
10 20 30

     // check the )scripts list
     )scripts
sc
     // the script 'sc' is now in the list.
```

)off

Clears all variables, functions, etc, from the active Cielo Explorer. Also closes the current open Solution in Visual Studio.

Syntax:

```
)off
```

Remarks:

The *off* command has the same effect as the *clear* command for the contents of the session, and also closes the currently open Solution in Visual Studio, and all associated projects.

If any open files are currently marked as unsaved in Visual Studio, then the Save File dialog is opened prompting for user action. This behavior is the built-in functionallity of Visual Studio, meaning that in relation to the currently open Solution, the off command has the same effect as the "File > Close Solution" menu item.

Example:

```
     a = 10 20 30
     a
10 20 30

     )off

     a
The variable 'a' does not exist
```

)run

Executes the contents of a script from the current Session Project.

Syntax:

```
)run script
```

script: The name of the script file to run.

Remarks:

The *)run* command looks in the *)scripts* list for a script named the specified name. If the script file exists in the list, the contents of the script are executed in the session.

The *)run* command is similar to the *)load* command, except that the specified script must already be listed in the *)scripts* list for the command to succeed.

Example:

```
     // check if 'a' exists
     a
name 'a' is not defined

     // check the contents of the )script list
     )scripts
sc
     // the 'sc' script is present in the session

     // run the script 'sc', which defines 'a'
     )run sc

     // check if 'a' exists
     a
10 20 30
```

)runf

Executes the contents of a script at the specified file path.

Syntax:

```
)runf scriptPath
```

scriptPath: The fully qualified file path of the script to run.

Remarks:

The *)runf* command takes a file path to a script file as its argument. When the *)runf* command is entered, the contents of the specified script file are run in the session.

The *)runf* command is similar to the *)run* command, except that the argument script file does not need to exist in the *)scripts* list.

Example:

```
     // check if 'a' exists
     a
name 'a' is not defined

     // check the contents of the )script list
     )scripts

     // the list is empty

     // runf the script 'sc', which defines 'a'
     )runf c:\sc.apl

     // check if 'a' exists
     a
10 20 30
```

)scripts

Shows a list of scripts which are currently loaded into the session.

Syntax:

```
)scripts
```

Remarks:

Script files can be loaded into the session by the *)load*, *)xload*, and *)edit* commands. Pressing **Ctrl+E+E** in an open script file also adds that script to the *)scripts* list.

Example:

```
     // check the contents of the )script list
     )scripts
sc1 sc2 sc3
     // there are three scripts currently loaded

     // xload a script called 'math'
     )xload math

     // check the )scripts list
     )scripts
sc1 sc2 sc3 math
     // the script 'math' is now in the list.
```

)vars

Shows the current list of variables which have been defined in the session.

Syntax:

```
)vars
```

Remarks:

The *)vars* command shows the list of variables which have been created in the session.

Example:

```
     // check the contents of the )vars list
     )vars
a b
     // there are two variables currently defined

     // define a new variable called 'c'
     c = 100 200 300

     // check the )vars list
     )vars
a b c
     // the variable 'c' now exists in the session.
```

)xload

Loads a script from the current Session Project.

Syntax:

```
)xload script
```

script: The name of the script file to xload.

Remarks:

The *)xload* command looks in the current Session Project for a script named the specified name. If the script file exists, it is added to the *)scripts* list in the session.

This command has a similar behavior to the *)load* command, except that the contents of the script are not executed.

Example:

```
     // check the contents of the )script list
     )scripts

     // the )scripts list is empty

     // xload a script called 'sc'
     )xload sc

     // check the )scripts list
     )scripts
sc
     // the script 'sc' is now in the list.
```

)xmlout

Exports a variable in XML format into the current Session Project in Visual Studio.

Syntax:

```
)xmlout var var var...
```

var: The name of a variable to export.

Remarks:

For each argument variable, the *xmlout* command creates an XML file in the current Session Project named "*var*.xml", where *var* is the name of the variable being exported.

If an XML file by that name already exists in the current Session Project, then that file is overwritten with the newly produced XML output.

Only objects which are serializable can be successfully exported to XML.

A variable is considered serializable if any of the following conditions are met:

- It is marked with the .Net Serializable attribute.
- Implements the .Net ISerializable interface.
- Implements the IXMLSerializable interface.
- Has a registered serializer in the Cielo Explorer.

If an object is encountered in a variable being exported with *xmlout* that does not meet any of the above serializable criteria, then a comment is placed in the generated XML at the location where the object would have appeared in the output XML, stating that the element could not be serialized. This behavior ensures that if you have a variable in the Cielo Explorer which contains mostly serializable data and only a few elements which cannot be serialized, the elements which are not serialzable will not prevent the serializable elements from being exported to XML format. Keep in mind that if you save the generated XML back to the Cielo Explorer by the use of **Ctrl+E+E**, that those elements which could not be serialized during the generation of the XML file will contain empty objects in the newly imported variable.

Once the *xmlout* command has been executed, the active window in Visual Studio is shifted to the newly created or updated XML file. If more than one variable was exported, the focus is placed on each XML file view as it is created, ultimately being placed on the XML file of the last variable being exported.

Saving changes to XML variables

Once a variable has been exported as XML, the generated XML can be modified in any way desired, and those changes can be saved back to the Cielo Explorer.

To save changes made to an XML file in the current Session Project, open that file and press the key sequence **Ctrl+E+E**

Once the sequence is pressed, focus is returned to the Cielo Explorer, and a message is printed to the session showing that an update was made to the variable.

When the changed XML is saved back to the session, the name of the variable into which the data is saved is taken from the name of the XML file. This means that if an XML file named "a.xml" is saved to the session using **Ctrl+E+E**, then the deserialized contents of that file will be saved as the variable "a" in the session. This means that you can not only save variables from the session in XML format, but you can also import entirely new variables from XML format by simply adding them to the Session Project, opening them, and

then pressing **Ctrl+E+E**

Additional Information

The *xmlout* command uses the XmlCvarSerialzier to perform the XML conversion to and from the current Session Project.

Example:

```
a = 10  test  (15)            "    "
)xmlout a
// a file has been created in the current Session Project.
```

Cielo Explorer Menu Reference

The Cielo Explorer includes a toolbar which allows various common session management activities to be easily performed at the click of a button.

Following is a listing of each button in the Cielo Explorer and their uses:

Toolbar buttons in Cielo Explorer

New

Clears the present session. This unloads the present domain, removing all references to assemblies and creates a new session domain.

Run Cielo Script

The user is prompted with the file selection dialog box. A script file is selected which is then defined and run in the current session. Scripts can contain any statement or expression; this includes control structures, function definitions, classes and simple statements. Scripts are dynamic, and functions, variables or classes defined in a script replace any dynamic members that exist in the current session with the same names.

For instance, the following script defines the function fn and then calls that function:

```
// Script: sc

function fn(a) {
          return a
}
fn( hello )
```

When this script is run in the session, the word hello is displayed in the session and the function fn is added to those available in the session.

Load Cielo File

This loads a Visual APL file which contains a formal assembly definition. The assembly is created and the resulting dll or exe can be referenced in the session or in the case of an exe, run from the OS.

Import Assembly

This adds a reference to an existing assembly to the session. If there is a namespace in the assembly which matches the name of the assembly, a using is also done.

Load Session Log

This prompts the user with a file selection dialog box. The user can choose any existing Visual APL log file, which is then displayed in the session, thus providing the user with all of the commands, definitions, etc which occurred in a previous session.

Save Session Log

This action saves all of the display content in the existing session to the file selected by the user. The user is prompted with the Save File dialog box.

Print

This prints the display contents of the existing session.

Cut

This removes the selected text from the session display and places it on the clipboard.

Copy

This copies the selected text which is placed on the clipboard.

Paste APL+Win

This pastes APL+Win code into the session explicitly converting from the legacy APL+Win text to APL Unicode.

:IF :ELSE

The tests for the :if and :elseif must evaluate to a single value which can be converted to a Boolean.

```
:if test
      if statement block
:elseif test1
      elseif statement block
:elseif test2
      elseif statement block
:else
      else statement block
:endif
```

The logical && and || are supported also.

In the example below the test2 is evaluated only if test returns a true

```
:if test && test2
      code block
:endif
```

In the example below the test2 is evaluated only if test returns a false

```
:if test || test2
      code block
:endif
```

:select :case

The :select control structure provides a mechanism for switching between multiple cases based on Identity comparison.

```
:select value
      :case value1
            code block
      :case value2
            code block
      :else
            else code block
:endselect
```

:while

The :continue keyword passes control to the :while test statement.

The :leave keyword branches to the first statement after the :while structure.

The test must return a value which will convert to Boolean

```
:while test
      statements
:endwhile
```

The logical && and || also works with the :while structure

In the example below the test2 is evaluated only if test returns a true

```
:while test && test2
      code block
:endwhile
```

In the example below the test2 is evaluated only if test returns a false

```
:while test || test2
      code block
:endwhile
```

:repeat :until

The :continue keyword passes control to the :until test statement.

The :leave keyword branches to the first statement after the :repeat structure.

The test must return a value which will convert to Boolean

The :repeat structure is repeated until the test evaluates to true.

```
:repeat
      code block
:until test
```

The logical && and || also works with the :repeat structure

In the example below the test2 is evaluated only if test returns a true

```
:repeat
      code block
:until test && test2
```

In the example below the test2 is evaluated only if test returns a false

```
:repeat
      code block
:until test || test2
```

:for :in

The :for control structure iterates across an iterable expression, placing the iterated values in the control variables.

```
:for i :in ⍳3
      print i
:endfor
0
1
2
```

The :continue keyword branches to the top of the for loop and the next value is assigned to the control variables.

The :leave keyword branches to the first statement after the :for loop.

The assignment of values into the variables follows the rules of variable assignment.

```
:for a b c :in (1 2 3) (4 5 6)
      print a
      print b
      print b
:endfor
```

The first time through the :for loop a:1, b:2, c:3 the second time a:4,b:5,c:6

It is also possible to assign based on depth of nested array

```
function fnf() {
    v = (1 (2 (3 4)) 5) (6 (7 (8 9)) 0)
    :for (a (b (c)) d) :in v
        print c
    :endfor
}
fnf()
3 4
8 9
```

: Label separator, switch case separator and legacy keyword indicator

Creates a label to which control can branch when used as follows:

```
function fn(a) {
      →L1
      print a
      L1:
      print  branch
}
```

Used to delimit legacy keywords

```
fuction fn(a) {
      :for i :in ι10
            print i
      :endfor
}
```

Used to delimit switch case statement

```
function fn(a) {
      switch (a) {
            case 10:
                  print  something
                  break
            default:
                  print  default
                  break
      }
}
```

→ Branch

The example below shows an unconditional branch to a label.

Example:

```
function fn(a) {
    print  one                    "   "
    →L1
    print  two                    "   "
    L1:
    print  three                  "     "
}
```

:goto :return

:goto provides an unconditional branch to a label

:goto label

:return returns from the function

It is also possible to return data with the :return keyword

:return expression

Using :return with an expression returns a value without having to set the default return variable specified in the user defined function header.

: Label separator, switch case separator and legacy keyword indicator

Creates a label to which control can branch when used as follows:

```
function fn(a) {
      →L1
      print a
      L1:
      print  branch
}
```

Used to delimit legacy keywords

```
fuction fn(a) {
      :for i :in ι10
            print i
      :endfor
}
```

Used to delimit switch case statement

```
function fn(a) {
      switch (a) {
            case 10:
                  print  something
                  break
            default:
                  print  default
                  break
      }
}
```

Number sign

Delimits directives, such as region.

```
#region code
      function fn(a) {
            print a
      }
#endregion
```

This creates a collapsible region in Visual Studio.

: Label separator, switch case separator and legacy keyword indicator

Creates a label to which control can branch when used as follows:

```
function fn(a) {
      →L1
      print a
      L1:
      print  branch
}
```

Used to delimit legacy keywords

```
fuction fn(a) {
      :for i :in ι10
            print i
      :endfor
}
```

Used to delimit switch case statement

```
function fn(a) {
      switch (a) {
            case 10:
                  print  something
                  break
            default:
                  print  default
                  break
      }
}
```

; Axis Separator

When used inside of an indexer bracket block [ ] the axis separator identifies the values for each axis.

```
      a = 1 2 3
      a[1]
2
      a = 3 3⍴⍳9
      a[1 2;1 2]
4 5
 7 8
```

It is not required to use the axis separator to index an array, for instance:

```
      b = (1 2) (1 2)
      a[b]
 4 5
 7 8
            b = 1 2
      a[b]
5
```

Providing a single value will index the array as though it were a vector.

```
       a[1]
1
```

You can select all values in an axis by using null:

```
      b = (1 2) (1 2) null
      a[b]
 12 13 14
 15 16 17

 21 22 23
 24 25 26
```

This makes it possible to index an array without having to be concerned about the syntax of the number of semi colons.

; Statement Separator

Separates code expressions or statements on a line.

Example:

```
      code1 ; code2
```

The diamond may also be used to delimit statements.

_ Underscore

A valid symbol to be used in a variable, method, function, property or other object name. It is also valid as the first character of a name.

In the session the _ contains the last information that was displayed to the screen or would have displayed to the screen if in a function.

Example:

```
      ⍳3
1 2 3
      _
1 2 3
```

This is very useful when reusing information in the session. Instead of having to copy and implement a long line of code, you can simply include _ on the next line.

Example:

```
       100+⍳10-3+5
100 101
     14+_
114 115
```

¯ High Minus

The high minus can be used to identify negative numbers in a vector or numbers being input.

For instance:

```
      10 - 5
5
```

However:

```
      10 ¯5
10  ¯5
```

This simplifies numeric input and reduces the need for parenthesis.

⍝ Comment

The ⍝ is the single line comment symbol. It can be used in conjunction with / to create a multi line comment.

Example:

```
        fn(a) {                  f
            b = a+1
            /⍝ this is a line
            another comment line
            yet another
            ⍝/
            print b
      }
      fn(10)
11
```

The double // also indicates a single comment line.

∇ Del

Delimiter used to identify the beginning of a user defined function.

Example:

```
∇ r←x add y {
      r←x+y
}
```

Notice that the beginning of the function block is started with a { and the end of the function block is terminated with a }.

∆ Delta

A valid symbol to be used in a variable, method, function, property or other object name. It is also valid as the first character of a name.

Note that objects that include the Δ will be difficult if not impossible to be consumed by other languages. This is included for legacy purposes.

≙ Delta underscore

A valid symbol to be used in a variable, method, function, property or other object name. It is also valid as the first character of a name.

Note that objects that include the ≙ will be difficult if not impossible to be consumed by other languages. This is included for legacy purposes.

◇ Statement Separator

Separates code expressions or statements on a line.

Example:

```
code1 ◇ code2
```

System Function Reference

This page contains a complete listing of all system quad functions currently available in Visual APL from APLNext.

Basic System Functions, Variables, etc.

System Function	Description
□DR	Data type and conversion
□ENLIST	Array to vector
□EXPAND	Array fill
□FI	Numeric format
□FIRST	First of an array
□FMT	Legacy Format
□FORMAT	New Array Formatter using .Net formatting specifiers
□MIX	Reduce nesting
□PENCLOSE	Array to nested vector
□REPL	Replicate array
□SPLIT	Increase nesting
□SS	String search
□TYPE	Numeric / character
□VI	Verify numeric
□FAVAIL	Returns a 1 if the share file system is available
□DM	Diagnostic message
□ERROR	Throw error
□dmx	Extended Diagnostic message
□DEF	Define function
□ERASE	Erase functions or variables
□EX	Erase functions or variables
□FX	Define function from □CR representation
□IDLIST	List objects in WS
□NC	List object types
□NL	List object names

□SIZE	Get size of object
□AT	Object attributes
□DL	Delay execution
□AI	Accounting information
□CT	Comparison tolerance
□IO	Index origin
□LIB	File directory
□LIBD	Set library to directory
□LIBS	List libraries and directories
□PP	Print precision
□RL	Random number seed
□TS	Timestamp
□reference	Adds a reference to an assembly
□using	Makes the namespace in a referenced assembly available
□AV	Atomic vector (character set)
□UCS	Returns index or Unicode character from index
□SYSID	APL system ID
□SYSVER	APL system version
□USERID	Workstation ID
□TCxx	Terminal control characters
□TC	contains a three-element vector of terminal control characters. □TC[1]= □TCBS (backspace) □TC[2]= □TCNL (newline) □TC[3]= □TCLF (linefeed).

Other Terminal Control Constants:

□TCBEL	Bell character
□TCBS	Backspace character
□TCDEL	Delete character
□TCESC	Escape character

□TCFF	Formfeed character
□TCHT	Horizontal Tab character
□TCLF	Linefeed character
□TCNL	Newline character
□TCNUL	Null character

State Functions

□ea	Executes either left or right arguments
□monadic	Indicates if an APL function was called monadically
□dyadic	Indicates if an APL function was called dyadically
□dbz	Divide By Zero
□dbzv	Divide By Zero Value
□nfi	NumberFormatInfo used by pattern format and when displaying to session

Argument Attributes

□arglist	Indicates argument is to used as list or arguments to the method
□argnames	Indicates argument is a matrix of named arguments and values

Application Shared DataStore (manages datastore created with svglobal keyword)

□svd	Remove a shared variable from the datastore
□svc	Check to see if a variable has been assigned since last assigned or referenced
□svs	Check to see if a variable is in the datastore
□svget	Sets an event method on a shared variable which runs when variable is referenced
□svset	Sets an event method on a shared variable which runs when variable is assigned

Windows Interface (legacy) loaded with Windows Interface Assembly

These quads have been deprecated in favor of the Windows Designer in Visual Studio and the new .Net System.Windows.Forms and related classes.

□wi	Windows Interface Legacy
□wself	The current or last reference wi object
□wres	Legacy wi wres
□warg	Legacy wi warg
□wsender	The actual object that created an event
□wievent	The actual event which was raised
□wevent	The legacy wi event

Loaded with the NativeFileSystem assembly

□NAPPEND	Add data to file
□NCREATE	Create file
□NERASE	Erase file
□NNAMES	Names of open files
□NNUMS	Numbers of open files
□NREAD	Read data
□NRENAME	Rename file
□NREPLACE	Replace data in file
□NRESIZE	Resize file
□NSIZE	Get file size
□NTIE	Tie (open) file
□NUNTIE	Untie file
□nexists	Deterimines if a file or directory exists
□ncopy	Copies a file to a new file
□nmove	Moves the file
□nstream	Returns he filestream associated with the tie number

Loaded with ShareFileSystem assembly

□FAPPEND	Append components
□FCREATE	Create file
□FDROP	Drops components from the beginning or end of a

	share file and renumbers the components
□FDUP	Duplicates a share file
□FERASE	Erase a share file
□FLIB	File directory
□FNAMES	Tied share file names
□FNUMS	Tied share file numbers
□FREAD	Read component
□FREPLACE	Replace component
□FSIZE	Get file size
□FSTIE	Tie share file
□FTIE	Tie share file
□FUNTIE	Untie share file
□fcatenate	Catenate a valuetype to a valuetype array stored in a component
□libdrw	Determines access to virtual share file directory
□libdcws	Changes access to virtual share file directory
□firead	Reads a specified range of valuetypes from a valuetype array in a component
□fireplace	Replaces a specified range of valuetypes in a valuetype array in a component
□falloc	Allocates contiguous space to a share file component
□fcnloc	Returns the physical location of a component in a share file
□fstream	Returns the filestream associated with the file tie number or virtual directory
□fremove	Removes a component from a share file and renumbers components

Loaded with either the Native File System or Share File System

□XLIB	Returns the directory or files in a directory
□CHDIR	Change current directory
□MKDIR	Create directory
□RMDIR	Delete directory

System Function Reference

This page contains a complete listing of all system quad functions currently available in Visual APL from APLNext.

Basic System Functions, Variables, etc.

System Function	Description
□DR	Data type and conversion
□ENLIST	Array to vector
□EXPAND	Array fill
□FI	Numeric format
□FIRST	First of an array
□FMT	Legacy Format
□FORMAT	New Array Formatter using .Net formatting specifiers
□MIX	Reduce nesting
□PENCLOSE	Array to nested vector
□REPL	Replicate array
□SPLIT	Increase nesting
□SS	String search
□TYPE	Numeric / character
□VI	Verify numeric
□FAVAIL	Returns a 1 if the share file system is available
□DM	Diagnostic message
□ERROR	Throw error
□dmx	Extended Diagnostic message
□DEF	Define function
□ERASE	Erase functions or variables
□EX	Erase functions or variables
□FX	Define function from □CR representation
□IDLIST	List objects in WS
□NC	List object types
□NL	List object names

□SIZE	Get size of object
□AT	Object attributes
□DL	Delay execution
□AI	Accounting information
□CT	Comparison tolerance
□IO	Index origin
□LIB	File directory
□LIBD	Set library to directory
□LIBS	List libraries and directories
□PP	Print precision
□RL	Random number seed
□TS	Timestamp
□reference	Adds a reference to an assembly
□using	Makes the namespace in a referenced assembly available
□AV	Atomic vector (character set)
□UCS	Returns index or Unicode character from index
□SYSID	APL system ID
□SYSVER	APL system version
□USERID	Workstation ID
□TCxx	Terminal control characters
□TC	contains a three-element vector of terminal control characters. □TC[1]= □TCBS (backspace) □TC[2]= □TCNL (newline) □TC[3]= □TCLF (linefeed).

Other Terminal Control Constants:

□TCBEL	Bell character
□TCBS	Backspace character
□TCDEL	Delete character
□TCESC	Escape character

□TCFF	Formfeed character
□TCHT	Horizontal Tab character
□TCLF	Linefeed character
□TCNL	Newline character
□TCNUL	Null character

State Functions

□ea	Executes either left or right arguments
□monadic	Indicates if an APL function was called monadically
□dyadic	Indicates if an APL function was called dyadically
□dbz	Divide By Zero
□dbzv	Divide By Zero Value
□nfi	NumberFormatInfo used by pattern format and when displaying to session

Argument Attributes

□arglist	Indicates argument is to used as list or arguments to the method
□argnames	Indicates argument is a matrix of named arguments and values

Application Shared DataStore (manages datastore created with svglobal keyword)

□svd	Remove a shared variable from the datastore
□svc	Check to see if a variable has been assigned since last assigned or referenced
□svs	Check to see if a variable is in the datastore
□svget	Sets an event method on a shared variable which runs when variable is referenced
□svset	Sets an event method on a shared variable which runs when variable is assigned

Windows Interface (legacy) loaded with Windows Interface Assembly

These quads have been deprecated in favor of the Windows Designer in Visual Studio and the new .Net System.Windows.Forms and related classes.

□wi	Windows Interface Legacy
□wself	The current or last reference wi object
□wres	Legacy wi wres
□warg	Legacy wi warg
□wsender	The actual object that created an event
□wievent	The actual event which was raised
□wevent	The legacy wi event

Loaded with the NativeFileSystem assembly

□NAPPEND	Add data to file
□NCREATE	Create file
□NERASE	Erase file
□NNAMES	Names of open files
□NNUMS	Numbers of open files
□NREAD	Read data
□NRENAME	Rename file
□NREPLACE	Replace data in file
□NRESIZE	Resize file
□NSIZE	Get file size
□NTIE	Tie (open) file
□NUNTIE	Untie file
□nexists	Deterimines if a file or directory exists
□ncopy	Copies a file to a new file
□nmove	Moves the file
□nstream	Returns he filestream associated with the tie number

Loaded with ShareFileSystem assembly

□FAPPEND	Append components
□FCREATE	Create file
□FDROP	Drops components from the beginning or end of a

	share file and renumbers the components
□FDUP	Duplicates a share file
□FERASE	Erase a share file
□FLIB	File directory
□FNAMES	Tied share file names
□FNUMS	Tied share file numbers
□FREAD	Read component
□FREPLACE	Replace component
□FSIZE	Get file size
□FSTIE	Tie share file
□FTIE	Tie share file
□FUNTIE	Untie share file
□fcatenate	Catenate a valuetype to a valuetype array stored in a component
□libdrw	Determines access to virtual share file directory
□libdcws	Changes access to virtual share file directory
□firead	Reads a specified range of valuetypes from a valuetype array in a component
□fireplace	Replaces a specified range of valuetypes in a valuetype array in a component
□falloc	Allocates contiguous space to a share file component
□fcnloc	Returns the physical location of a component in a share file
□fstream	Returns the filestream associated with the file tie number or virtual directory
□fremove	Removes a component from a share file and renumbers components

Loaded with either the Native File System or Share File System

□XLIB	Returns the directory or files in a directory
□CHDIR	Change current directory
□MKDIR	Create directory
□RMDIR	Delete directory

□ai Account Information

Legacy account information. Returns a four element vector, the second element of which is the time in milliseconds since the first time that □ai was referenced.

This is particularly useful when doing simple timing tests:

```
□io=1
ts = □ai[1]
for (I = 0;i<10000;i++) {
      b = 10×i
}
print □ai[1]-ts
```

This will display the time taken by the statements interposing the two references to □ai

The first element is always 1 and the last two elements are reserved.

□av Atomic Vector

This is provided for legacy reasons only.

Contains 256 characters and is a simple character vector. Visual APL is based on Unicode characters. □a v is a selection of commonly used Unicode characters.

cmd Command Window

This has been deprecated in favor of the System.Diagnostics.Process class.

Here is a simple example of how to use this:

```
using System.Diagnostics
a = Process()
a.StartInfo.FileName=  cmd.exe                "       "
a.StartInfo.UseShellExecute = false
a.StartInfo.Arguments =  /k dir *.*             "          "
a.Start()
```

This will open a cmd window and display the directory.

There are a wealth of options for this type and extensive documentation can be found for this .Net framework type at Microsoft.com, as well as the over 4,000 other .Net framework types.

⎕ct Comparison Tolerance

The comparison tolerance is the difference or fuzz allowed between two values when comparing them for equality. The default setting for ⎕ct is double.Epsilon which is the chip dependent comparison tolerance.

Example:

```
      using System
      double.Epsilon
4.94065645841247E-324
       ⎕ct
4.94065645841247E-324
```

The value of ⎕ct can be set to alter the operation of the following operators.

```
⌊ floor
⍳ index of
⌈ ceiling
> ≥ ≈ ≤ <  numeric relation
∣ residue
∈ find
≡ match
~ without
⍳ membership
```

Note
the ≈, or approximately equal symbol is obtained by pressing the alt-5 key. This is not to be confused with the = symbol which is used for reference assignment. To perform an exact equal use == `a == b`

⎕dr

The data representation of intrinsic objects in .Net can be determined and manipulated using ⎕dr.

⎕dr can be used either monadically or dyadically.

Monadic:
When used monadically ⎕dr reports the type of an object based on legacy codes. These codes are:

Code	Description
11:	boolean (true/false, not bit)
81:	bytes
82:	chars (compatible with 82 in existing system)
83:	reserved.
162:	chars (compatible with 82 in existing system)
163:	short (Int16, 16 bit integer)
164:	ushort (UInt16, unsigned short)
323:	int (Int32, 32 bit integer, default)
324:	uint (UInt32, unsigned int)
325:	float (Single, 32 bit real)
643:	long (Int64, 64 bit integer)
644:	ulong (UInt64, unsigned long)
645:	double (Double, 64 bit real, default)
1285:	Decimal (128 bit real)
807:	object (serialized object)
99999:	no code available for data type

Example:

```
      ⎕dr 10
323
    ⎕dr 10L
643
    ⎕dr 20.1
645
    ⎕dr 10f
325
```

Dyadic:

The left argument to ⍠dr can be a legacy code listed above. When this is the case the data on the right is coerced to the new data type based on the bit representation of the data.

Example:

Converts a short to a Boolean representation:

```
      11 ⍠dr (short)32
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

      323 ⍠dr 10.1
858993459 1076114227

      645 ⍠dr 323 ⍠dr 10.1
10.1

      645 ⍠dr (bool) 11 ⍠dr 10.1
10.1
```

Note

Requirement for casting to Boolean as the result of 11 ⍠dr 10.1 is an integer array of 1 and 0.

Often what is desired is to cast an int to a double, a double to an int, a short to and int, etc.

Using a type as the left argument to ⍠dr accomplishes this.

Example:

```
      int ⍠dr 10.1
10
      int ⍠dr 10.1 10.6
10 11
      ⍠dr double ⍠dr 10
645
```

It is also possible to serialize data using ⍠dr. This is accomplished using the text string "wrapl" as the left argument. To deserialize the data, use "unwrapl"

```
      a =  wrapl  ⍠dr 10  test  20'     "        "    "
      b =  wrapl  ⍠dr 10  test  20'     "        "    "
```

The result of the serialization is a string and the result is always identical for identical data. This means that the results can be compared for the purposes of checking equivalence.

Any object that supports serialization can be serialized either individually or as part of a nested structure.

If you understand the serialization of the object, you can even modify the string which will impact the object that you return.

This is useful for sending objects either over the internet or writing and object to file and then retrieving and reinstantiating the object at a later time.

⎕dbz Divide By Zero

This system function provides control over the way in which the system addresses divide by zero.

The default value is 0 to match .Net languages, however, you can set this to the following:

```
⎕dbz:
 0 :   1 0 = 0                                  ÷
       0 0 = 0                                  ÷
 1 :   1 0 = DOMAIN ERROR                       ÷
       0 0 = 1                                  ÷
 2 :   1 0 = DOMAIN ERROR                       ÷
       0 0 = DOMAIN ERROR                       ÷
 3 :   1 0 = NaN or ⎕dbzv                       ÷
       0 0 = NaN or ⎕dbzv                       ÷
 4 :   1 0 = +-Infinity                         ÷
       0 0 = NaN                                ÷
```

You can set ⎕dbzv to any object, and that will be returned when ⎕dbz is set to 3

There are several new Double types which are valid doubles and therefore do not promote a double array to a heterogeneous array.

```
      double.NaN
NaN
      double.NegativeInfinity
-Infinity
      double.PositiveInfinity
Infinity

      ⎕dbzv← byzero                "      "
      ⎕dbz←3
      a←2 3⍴⍳6
      a 2 3⍴10 0                                ÷
         0     byzero    0.2
    byzero        0.4 byzero

      ⎕dbzv←double.NaN
      a 2 3⍴10 0                                ÷
       0 NaN     0.2
 NaN     0.4 NaN
      b = a 2 3⍴10 0                                ÷
      ⎕dr b
645
```

⎕dyadic

Indicates if a user defined function was called with both a left and right argument. ⎕dyadic is false if the function was called with only the right argument.

```
∇r←a add b {
      if (⎕dyadic) {
            r←a+b
      } else {
            r←b
      }
}
```

Dynamically Referencing Assemblies

The **refbyfile**, **refbyname**, and **using** keywords are directives and are only referenced during creation of the dll or exe assembly.

For late binding to an assembly, Visual APL supplies two quad system functions:

□**reference**

□**using**

□reference adds a late bound reference to the specified assembly, whether it is given as a file or as a name, and does this during execution. Arguments to □reference can be any valid APL expression which produces a string. For instance:

```
a = @ c:\myprojects\myutils".dll                    "
 □reference a
```

Or to reference by name:

```
□reference  System.Windows.Forms"                    "
```

Both will return true if successful and false if it fails to load the assembly.

Once an assembly has been loaded, you can then use the namespaces in that assembly, for example:

```
□using  myutils             "       "
 □using  System.Windows.Forms"                    "
```

You can also specify an alias for a using like this:

```
□using win =  System.Windows.Forms"                    "
```

The variable win will now contain the System.Windows.Forms assembly information.

Aliases are used to avoid name conflicts between assemblies.

As these are evaluated during execution, any valid APL expression can create the input to these system quad functions.

However, if you are using an alias it must be the first assignment in the expression before the □using.

□ex Expunge

Erases a global object or sets a local object to it's default value. Returns a 1 if successful, or a 0 if the object could not be erased or set to its default value.

Local variables that are dynamic are set to the default value for the data type which they contain when □ex is run on them. If they contain a ValueType they are set to the default for the particular value type, otherwise they are set to null.

Local variables that are strong typed are set to the default value for the data type which they must always contain. If a local variable is typed to int, then the erase will always set the value to 0, a Boolean type is set to false, etc. If a local typed variable is typed to a non ValueType then the value is set to null.

Setting a local variable to null will cause the garbage collector to remove the object to which the variable referred.

Erasing a global object removes the pointer to the object from the global dictionary and the object referenced is removed at the next garbage collection.

```
      a = 10
      b = 10 20 30
      □ex   a   b                  " " " "
1 1
```

The .Net framework documentation has a large section on garbage collection and the garbage collection class is available on System.GC

You should read the documentation and examples available from Microsoft very carefully before using GC.

⎕io Index Origin

This is the index origin that the operators will use for indexing and numbering.

For instance, setting ⎕io to 0:

```
      ⍳10
0 1 2 3 4 5 6 7 8 9
      ⎕io←1
      ⍳10
1 2 3 4 5 6 7 8 9 10
```

Conversely, indexing with ⎕io set to 0, which is the default for .Net languages results as follows:

```
      a = 1 2 3 4 5
      a[1]
2
      ⎕io←1
      a[1]
1
```

Note that when a type has an indexer you must honor the ⎕io of that type. Setting ⎕io will always affect the operators and indexing of arrays, however, specific types with indexers will still have their own internal origin which must be honored.

⎕io is local to the class. There is a ⎕io for each instance of a class and also the static version.

□monadic

Indicates whether the user defined function was called with a left argument or not. □monadic is true if the function was called without a left argument.

```
▿r←a add b {
      if (monadic) {
            r←b
      } else {
            r←a+b
      }
}

      add 10
10
      10 add 20
20
```

⎕nc Name Class

Monadic:
Returns a vector of integers indicating the type of object identified within a string as the right argument. The valid identifiers are:

Identifier	Meaning
0:	Does not exist in present scope
2:	Variable, Field or property
3:	Function or method
4:	Other, most likely a class

Example:

```
      ⎕nc  a b c
2 3 0
```

This would indicate that a is a variable, b is a function and c does not exist.

One of the most common uses for ⎕nc is to identify if a left argument has been passed to a user defined function. See ⎕monadic to simplify and speed up this test.

```
∇ r←a add b {
      if (0 == ⎕nc  a ) {
            r←b
      } else {
            r←a+b
      }
}


∇ r←a add b {
      if (⎕monadic) {
            r←b
      } else {
            r←a+b
      }
}
```

□nl Name List

Returns a string array of objects that match the following numeric identifiers:

identifier	object type
2:	variable, property or field
3:	function or method

Example:

```
      □nl 2
a   b
      □nl 3
  fn
      □nl 2 3
a   b   fn
```

□nfi

□nfi provides the instance of the NumberFormatInfo class which is used by □fmt and pattern format (₸)
Changes the properties of this object are reflected in the subsequent formatting output.

```
      nfi = □nfi
     nfi.NegativeSign =  -                          " "
       N2  □fmt ¯10            "   "
-10.00                  "       "
       N2  □fmt ¯10 ¯20.5      "   "
-10.00    -20.50        "       " "       "
```

This description by Microsoft of the way the NumberFormatInfo class is defined provides a rather complete layout of the different properties which can be set.

The values available on the NumberFormatInfo class are determined by the regional and culture settings of the computer.

There are additional members of the NumberFormatInfo class which are revealed either on the intellisense or in the detailed .Net framework information from Microsoft.

NumberFormatInfo Class

Defines how numeric values are formatted and displayed, depending on the culture.

Namespace: System.Globalization
Assembly: mscorlib (in mscorlib.dll)

This class contains information, such as currency, decimal separators, and other numeric symbols.

To create a **NumberFormatInfo** for a specific culture, create a CultureInfo for that culture and retrieve the CultureInfo.NumberFormat property. To create a **NumberFormatInfo** for the culture of the current thread, use the CurrentInfo property. To create a **NumberFormatInfo** for the invariant culture, use the InvariantInfo property for a read-only version, or use the **NumberFormatInfo** constructor for a writable version. It is not possible to create a **NumberFormatInfo** for a neutral culture.

The user might choose to override some of the values associated with the current culture of Windows through Regional and Language Options (or Regional Options or Regional Settings) in Control Panel. For example, the user might choose to display the date in a different format or to use a currency other than the default for the culture. If the CultureInfo.UseUserOverride property is set to **true**, the properties of the CultureInfo.DateTimeFormat instance, the **CultureInfo.NumberFormat** instance, and the CultureInfo.TextInfo instance are also retrieved from the user settings. If the user settings are incompatible with the culture associated with the **CultureInfo** (for example, if the selected calendar is not one of the OptionalCalendars), the results of the methods and the values of the properties are undefined.

Before .NET Framework version 2.0, if the **CultureInfo.UseUserOverride** property is set to **true**, then the object reads each user-overridable property only when it is accessed for the first time. Because **NumberFormatInfo** has more than one user-overridable property, that "lazy initialization" can lead to an inconsistency between such properties when the following occurs: the application accesses one property; then the user changes to another culture or overrides properties of the current user culture through Regional and Language Options in OS Control Panel; then the application accesses a different property. For example, in a sequence like this, CurrencyGroupSeparator could be accessed; then the user could change patterns in OS control panel, and CurrencyDecimalSeparator, when accessed, would follow the new settings. Similar inconsistency will happen when user change user culture in OS control panel.

In .NET Framework version 2.0 and later, **NumberFormatInfo** does not use this "lazy initialization". Instead, it reads all user-overridable properties when it is created. There is still a tiny window of vulnerability (neither

object creation nor the user override process is atomic, so the relevant values could change in the midst of object creation), but this should be extremely rare.

Numeric values are formatted using standard or custom patterns stored in the properties of a **NumberFormatInfo**. To modify how a value is displayed, the **NumberFormatInfo** must be writable so custom patterns can be saved in its properties.

The following table lists the standard format characters for each standard pattern and the associated **NumberFormatInfo** property that can be set to modify the standard pattern.

Format Character	Description and Associated Properties
c, C	Currency format. CurrencyNegativePattern, CurrencyPositivePattern, CurrencySymbol, CurrencyGroupSizes, **CurrencyGroupSeparator**, CurrencyDecimalDigits, **CurrencyDecimalSeparator**.
d, D	Decimal format.
e, E	Scientific (exponential) format.
f, F	Fixed-point format.
g, G	General format.
n, N	Number format. NumberNegativePattern, NumberGroupSizes, NumberGroupSeparator, NumberDecimalDigits, NumberDecimalSeparator.
r, R	Roundtrip format, which ensures that floating point numbers converted to strings will have the same value when they are converted back to numbers.
x, X	Hexadecimal format.

For details about these patterns, see Standard Numeric Format Strings and Custom Numeric Format Strings.

A DateTimeFormatInfo or a **NumberFormatInfo** can be created only for the invariant culture or for specific cultures, not for neutral cultures. For more information about the invariant culture, specific cultures, and neutral cultures, see the **CultureInfo** class.

This class implements the ICloneable interface to enable duplication of **NumberFormatInfo** objects. It also implements IFormatProvider to supply formatting information to applications.

□ print string representation

The □ does not take input from the keyboard. This is handled with streams in .Net. However, the □ is used to print data to the session. In particular, evaluated expressions do not produce output to the screen inside of a function. Using □ explicitly prints output to the session using the string representation of the object.

```
      function fn(a) {
            □←a+10
            a+10
      }
      fn(10)
20
```

The print keyword performs the same action:

```
      function fn(a) {
            print a+10
            a+10
      }
      fn(10)
20
```

□rl Random Link

The .Net framework provides a random number generator and the details of the generator can be found in the .Net framework documentation from Microsoft.

The roll and deal operations rely on `□rl`, which is the random link.

The default value for `□rl` is `16807`. However, the sequence of random numbers generated will be based on the random algorithm in the .Net framework.

Example:

```
      □rl
16807
      □rl←1230303
      □rl
1230303
      ?10
2
      ?10
9
      ?10
10
      □rl←1230303
      ?10
2
      ?10
9
      ?10
10
      □rl
872203611
```

⎕sysid System Identification

Returns a string with the name of the language.

```
      ⎕sysid
Visual APL for Windows

      or
      ⎕sysid
Visual APL for Linux

      or
      ⎕sysid
Visual APL for Macintosh
```

sysver System Version

Returns a string containing the information about the current build of the language.

```
        sysver
1.0.2400  on .Net 2.0.50727.42
```

⎕fi

Converts a string or character array to numeric data. Blanks are considered as delimiters and 0 is used to replace ill formed numbers.

```
      ⎕fi '3.6 2E2 ,1 THREE 0'
3.6 200 0 0 0

      ⎕fi '6.25 ¯6.25'
6.25 ¯6.25
```

Notice that the negative is shown as a middle minus. This is because the result of ⎕fi in this case is a native double vector.

If you use ravel you will see:

```
      ,⎕fi '6.25 ¯6.25'
6.25 ¯6.25
```

Which is the display for a Visual APL data type, which is created when the data is raveled. This can be cast back to native double by simply:

```
      (double) ,⎕fi '6.25 ¯6.25'
6.25 ¯6.25
```

In which case the data is now a native double again.

It is not required to use only a string with ⎕fi. You can use several strings or numbers.

```
        ⎕fi 10
10
       ⎕fi 10  10 10 10   100        "        " "   "
10 10 10 10 100
```

This reduces the cost of catenation and concern about data types as the input to ⎕fi.

⎕ts TimeStamp

Returns the current time stamp in a seven-element integer vector consisting of the year, month, day, hour, minute, second, and millisecond.

```
    ⎕ts
2006 7 28 18 42 2 304
```

This has been largely deprecated with the DateTime object in .Net

```
      using System
      DateTime.Now
7/28/2006 6:43:18 PM
      a = DateTime.Now
```

There are innumerable properties and methods on both the DateTime class and the instance of the DateTime.Now reference. In addition there are a wide range of formatters available for the DateTime class. See ⎕fmt for use of the DateTime format information.

It is also simple to do comparisons of time:

```
       DateTime.Subtract(DateTime.Now,a)
00:50:04.2198560
```

The DateTime.Subtract method returns a TimeSpan object which has numerous methods and properties which makes the analysis of the time difference very simple.

⎕ucs Universal Character Set

Translates between integers and Unicode characters.

Example:

```
      ⎕ucs  a←⍳10                    "      "
97 8592 9075 49 48
     ⎕ucs ⎕ucs  a←⍳10                  "      "
a ← ⍳ 1 0
```

If the right argument is a string or characters integers are returned

□userid User ID

Returns the name of the machine on which the system is running.

```
      □userid
workstation12
```

This has been deprecated in favor of the System.Environment object.

□vi

Returns an array of 1's and 0's which represent if the data, delimited by blanks, is a well formed number representation or not.

```
      □vi '3.6 2E2 ,1 THREE 0'
1 1 0 0 1
```

□vi also takes multiple strings or numeric data as an argument.

```
      □vi 10  10 10 10   100        "        " "   "
1 1 1 1 1
```

□format

□format uses all of the intrinsic .Net formatting and also includes control of widths, for all array sizes, for instance:

```
     10  N2  □Format 23.34      "  "
     23.34
```

□format makes it possible to apply a format specifier across an array or singleton. It also adds the ability to specify width of format, as shown above.

For Example:

```
       10  N2  □format 2 2⍴10 11  "  "
     10.00     11.00
     10.00     11.00
```

Without width specified:

```
       N2  □format 2 2⍴10 1111'1.1' 30.4
10.00 11,111.10
30.40 10.00
```

Notice that there are no pre set widths for the columns. This has the advantage of not losing data when formatting, but the disadvantage of not being able to control column widths.

For example:

```
        7  N2  □format 2 2⍴10 1234'5.2' 30.5
  10.00*******
  30.50  10.00

        N2  □format 2 2⍴10 1234'5.2' 30.5
10.00 12,345.20
30.50 10.00

Format can be applied by column:

       N2   C2  □format 2 2⍴10' 20' 3'0 4'0
10.00 $20.00
30.00 $40.00
```

If there are more columns than format strings, then the string are reapplied in column order:

```
       N2   C2  □format 2 4⍴10' 20' 3'0 4'0
10.00 $20.00 30.00 $40.00
10.00 $20.00 30.00 $40.00
```

The same applies for column widths and formats:

```
      7  N2  10  C2  □format 2 4'⍴10' 20 3'0 4'0
  10.00    $20.00  30.00    $40.00
  10.00    $20.00  30.00    $40.00
```

If column widths are specified, they must be specified for all columns.

The formats can also be specified for each element in the array:

```
      a = (2 4⍴ N2   C2   C3   N3   C4 "  N3 "  N5 "  C6 ")  " "  " "  " "  " "  "
```

```
      a ⎕format 2 4⍴10 20 30 40
10.00    $20.00 $30.000  40.000
$10.0000 20.000 30.00000 $40.000000
```

The formats for each element in the array can also contain width settings:

```
      a = (2 4⍴(7  N2 ) (8  C2 ) (7  C3 ))"  "     "  "     "  "
      a ⎕format 2 4⍴10 20 30 40
 10.00  $20.00$30.000  40.00
 10.00  $20.00$30.000  40.00
```

In .Net an object can contain its own format information. The DateTime object contains its own format information. With ⎕format you can apply the formatting to an object in an array, DateTime.Now returns an object with the current time information. We can format it like this:

```
       d  ⎕format DateTime.Now⍵ "
7/27/2006

      F  ⎕format DateTime.Now⍵ "
Thursday, July 27, 2006 12:33:47 PM
```

These can be applied using ⎕format to an array:

```
      d   N2   F  ⎕format 2 3⍴⍵⍵'DateTime'.Now 100 DateTime.Now
 7/27/2006  100.00  Thursday, July 27, 2006 12:34:54 PM
 7/27/2006  100.00  Thursday, July 27, 2006 12:34:54 PM
```

These formatting concepts apply to all objects in the .Net framework or objects created which contain their own formatting information.

One of the difficult problems with formatting is addressing comma delimiter by region, the high minus and other issues. These can be set using ⎕nfi.

For instance:

```
      nfi = ⎕nfi
      nfi.NegativeSign = -                    " "
       N2  ⎕format ¯10       "  "
-10.00               "      "
      N2  ⎕format ¯10 ¯20.5 "  "
-10.00   -20.50       "      " "      "
```

This shows changing the high minus to a middle minus. There are many regional and culture specific formatting options which are available to be set, which are shown in the documentation or with intellisense.

Setting regional setting will also affect the formatting. This means that when your formatting is performed on a machine with a different culture set, the correct currency, command and period delimiters will be used. Of course as we have shown these can be specifically overridden using ⎕nfi.

The following outlines how to use each of the formatting specifiers. These can be used with ⎕format or uniquely on a single scalar as shown below.

For additional information on the .Net formatting structure as provided by Microsoft see the related sections in this help or the Microsoft online help.

Composite Formatting

The .NET Framework composite formatting feature takes a list of objects and a composite format string as input. A composite format string consists of fixed text intermixed with indexed placeholders, called format items, that correspond to the objects in the list. The formatting operation yields a result string that consists of the original fixed text intermixed with the string representation of the objects in the list.

The composite formatting feature is supported by methods such as Format, AppendFormat, and some overloads of WriteLine and TextWriter.WriteLine. The String.Format method yields a formatted result string, the **AppendFormat** method appends a formatted result string to a StringBuilder object, the Console.WriteLine methods display the formatted result string to the console, and the TextWriter.WriteLine method writes the formatted result string to a stream or file.

Composite Format String

A composite format string and object list are used as arguments of methods that support the composite formatting feature. A composite format string consists of zero or more runs of fixed text intermixed with one or more format items. The fixed text is any string that you choose, and each format item corresponds to an object or boxed structure in the list. The composite formatting feature returns a new result string where each format item is replaced by the string representation of the corresponding object in the list.

Consider the following **Format** code fragment.

```
Visual APL
myName =  Davin ;                  "      "
 String.Format( Name = {0}, hours = {1:hh} , myName, DateTime.Now);
```

The fixed text is "Name = " and ", hours = ". The format items are "{0}", whose index is 0, which corresponds to the object myName, and "{1:hh}", whose index is 1, which corresponds to the object DateTime.Now.

Format Item Syntax

Each format item takes the following form and consists of the following components:

{index[**,**alignment][**:**formatString]**}**

The matching braces ("{" and "}") are required.

Index Component

The mandatory *index* component, also called a parameter specifier, is a number starting from 0 that identifies a corresponding item in the list of objects. That is, the format item whose parameter specifier is 0 formats the first object in the list, the format item whose parameter specifier is 1 formats the second object in the list, and so on.

Multiple format items can refer to the same element in the list of objects by specifying the same parameter specifier. For example, you can format the same numeric value in hexadecimal, scientific, and number format by specifying a composite format string like this: "{0:X} {0:E} {0:N}".

Each format item can refer to any object in the list. For example, if there are three objects, you can format the second, first, and third object by specifying a composite format string like this: "{1} {0} {2}". An object that is not referenced by a format item is ignored. A runtime exception results if a parameter specifier designates an item outside the bounds of the list of objects.

Alignment Component

The optional *alignment* component is a signed integer indicating the preferred formatted field width. If the value of *alignment* is less than the length of the formatted string, *alignment* is ignored and the length of the formatted string is used as the field width. The formatted data in the field is right-aligned if *alignment* is

positive and left-aligned if *alignment* is negative. If padding is necessary, white space is used. The comma is required if *alignment* is specified.

Format String Component

The optional *formatString* component is a format string that is appropriate for the type of object being formatted. Specify a numeric format string if the corresponding object is a numeric value, a date and time format string if the corresponding object is a DateTime object, or an enumeration format string if the corrersponding object is an enumeration value. If *formatString* is not specified, the general ("G") format specifier for a numeric, date and time, or enumeration type is used. The colon is required if *formatString* is specified.

Escaping Braces

Opening and closing braces are interpreted as starting and ending a format item. Consequently, you must use an escape sequence to display a literal opening brace or closing brace. Specify two opening braces ("{{") in the fixed text to display one opening brace ("{"), or two closing braces ("}}") to display one closing brace ("}"). Braces in a format item are interpreted sequentially in the order they are encountered. Interpreting nested braces is not supported.

The way escaped braces are interpreted can lead to unexpected results. For example, consider the format item "{{{0:D}}}", which is intended to display an opening brace, a numeric value formatted as a decimal number, and a closing brace. However, the format item is actually interpreted in the following manner:

1. The first two opening braces ("{{") are escaped and yield one opening brace.
2. The next three characters ("{0:") are interpreted as the start of a format item.
3. The next character ("D") would be interpreted as the Decimal standard numeric format specifier, but the next two escaped braces ("}}") yield a single brace. Because the resulting string ("D}") is not a standard numeric format specifier, the resulting string is interpreted as a custom format string that means display the literal string "D}".
4. The last brace ("}") is interpreted as the end of the format item.
5. The final result that is displayed is the literal string, "{D}". The numeric value that was to be formatted is not displayed.

One way to write your code to avoid misinterpreting escaped braces and format items is to format the braces and format item separately. That is, in the first format operation display a literal opening brace, in the next operation display the result of the format item, then in the final operation display a literal closing brace.

Processing Order

If the value to be formatted is **null** (**Nothing** in Visual Basic), an empty string ("") is returned.

If the type to be formatted implements the ICustomFormatter interface, the ICustomFormatter.Format method is called.

If the preceding step does not format the type, and the type implements the IFormattable interface, the IFormattable.ToString method is called.

If the preceding step does not format the type, the type's **ToString** method, which is inherited from the Object class, is called.

Alignment is applied after the preceding steps have been performed.

Code Examples

The following example shows one string created using composite formatting and another created using an object's **ToString** method. Both types of formatting produce equivalent results.

```
Visual APL
FormatString1 = String.Format( {0:dddd MMMM} , DateTime.Now);       "
FormatString2 = DateTime.Now.ToString( dddd MMMM );           "          "
```

Assuming that the current day is a Thursday in May, the value of both strings in the preceding example is Thursday May in the U.S. English culture.

The following example demonstrates formatting multiple objects, including formatting one object two different ways.

```
Visual APL
myName = Davin ; String.Format("Name = {0}, hours = {1:hh}, minutes =
{1:mm} , myName, DateTime.Now)";
```

The output from the preceding string is "Name = Fred, hours = 07, minutes = 23", where the current time reflects these numbers.

The following examples demonstrate the use of alignment in formatting. The arguments that are formatted are placed between vertical bar characters (|) to highlight the resulting alignment.

```
Visual APL
myFName = Davin ; string myLName = Opals ; int myInt = 100;     "
FormatFName = String.Format( First Name = |{0,10}| ," myFName);          "
FormatLName = String.Format( Last Name = |{0,10}| ,"myLName);           "
FormatPrice = String.Format( Price = |{0,10:C}| , myInt);              "
print String.Format(FormatFName);
print String.Format (FormatLName); Console.WriteLine(FormatPrice); FormatFName
= String.Format( First Name = |{0,-10}| ", myFName);           "
FormatLName = String.Format( Last Name = |{0,-10}| ," myLName); FormatPrice =
String.Format( Price = |{0,-10:C}| , myInt);              "
print String.Format(FormatFName);
print String.Format(FormatLName);
print String.Format(FormatPrice);
```

The preceding code displays the following to the console in the U.S. English culture. Different cultures display different currency symbols and separators.

```
First Name = | Davin|
Last Name = | Opals|
Price = | $100.00|
First Name = |Davin |
Last Name = | Opals |
Price = |$100.00 |
```

There is a great section on .Net formatting at:

http://msdn2.microsoft.com/en-us/library/dwhawy9k(VS.80).aspx

http://msdn2.microsoft.com/en-us/library/241ad66z(VS.80).aspx

Make sure to checkout the NumberFormatInfo object, which we reveal through nfi

For instance, if you do:

```
a = nfi
a.NegativeSign = -                          " "
```

Then formatting will use the middle minus for formatting instead of the high minus.

We also support the date and time formatting strings for a date/time object, which you can create with:

```
a = DateTime.Now
```

You can find this at:

http://msdn2.microsoft.com/en-us/library/az4se3k1(VS.80).aspx

http://msdn2.microsoft.com/en-us/library/hc4ky857(VS.80).aspx

You can place the DateTime object in a matrix and then when you format it will use the correct format, as:

```
              a = DateTime.Now 10.2
              'd' 'N2' format a
  6/15/2006  10.20
```

Standard DateTime Format Strings

A standard **DateTime** format string consists of a single standard **DateTime** format specifier character that represents a custom **DateTime** format stringCustom DateTime Format Strings

The format string ultimately defines the text representation of a DateTime object that is produced by a formatting operation. Note that any **DateTime** format string that contains more than one alphabetic character, including white space, is interpreted as a custom **DateTime** format string.

Standard DateTime Format Specifiers

The following table describes the standard **DateTime** format specifiers. For examples of the output produced by each format specifier, see Standard DateTime Format Strings Output Examples.

Format specifier	Name	Description
d	Short date pattern	Represents a custom **DateTime** format string defined by the current ShortDatePattern property. For example, the custom format string for the invariant culture is "MM/dd/yyyy".
D	Long date pattern	Represents a custom **DateTime** format string defined by the current LongDatePattern property. For example, the custom format string for the invariant culture is "dddd, dd MMMM yyyy".
f	Full date/time pattern (short time)	Represents a combination of the long date (D) and short time (t) patterns, separated by a space.
F	Full date/time pattern (long time)	Represents a custom **DateTime** format string defined by the current FullDateTimePattern property. For example, the custom format string for the invariant culture is "dddd, dd MMMM yyyy HH:mm:ss".
g	General date/time pattern (short time)	Represents a combination of the short date (d) and short time (t) patterns, separated by a space.
G	General date/time pattern (long time)	Represents a combination of the short date (d) and long time (T) patterns, separated by a space.
M or m	Month day pattern	Represents a custom **DateTime** format string defined by the current MonthDayPattern property. For example, the custom format string for the invariant culture is "MMMM dd".
o	Round-trip date/time pattern	Represents a custom **DateTime** format string using a pattern that preserves time zone information. The pattern is designed to round-trip **DateTime** formats, including the **Kind** property, in text. Then the formatted string can be parsed back using Parse or ParseExact with the correct **Kind** property value. The custom format string is "yyyy'-'MM'-'dd'T'HH':'mm':'ss.fffffffK". The pattern for this specifier is a defined standard. Therefore, it is always the same, regardless of the culture used or the format provider supplied.
R or r	RFC1123 pattern	Represents a custom **DateTime** format string defined by the current RFC1123Pattern property. The pattern is a defined standard and the property is read-only. Therefore, it is always the same regardless of the

		culture used or the format provider supplied. The custom format string is "ddd, dd MMM yyyy HH':'mm':'ss 'GMT'". Formatting does not modify the value of the **DateTime** object that is being formatted. Therefore, the application must convert the value to Coordinated Universal Time (UTC) before using this format specifier.
s	Sortable date/time pattern; conforms to ISO 8601	Represents a custom **DateTime** format string defined by the current SortableDateTimePattern property. This pattern is a defined standard and the property is read-only. Therefore, it is always the same regardless of the culture used or the format provider supplied. The custom format string is "yyyy'-'MM'-'dd'T'HH':'mm':'ss".
t	Short time pattern	Represents a custom **DateTime** format string defined by the current ShortTimePattern property. For example, the custom format string for the invariant culture is "HH:mm".
T	Long time pattern	Represents a custom **DateTime** format string defined by the current LongTimePattern property. For example, the custom format string for the invariant culture is "HH:mm:ss".
u	Universal sortable date/time pattern	Represents a custom **DateTime** format string defined by the current UniversalSortableDateTimePattern property. This pattern is a defined standard and the property is read-only. Therefore, it is always the same regardless of the culture used or the format provider supplied. The custom format string is "yyyy'-'MM'-'dd HH':'mm':'ss'Z'". No time zone conversion is done when the date and time is formatted. Therefore, the application must convert a local date and time to Coordinated Universal Time (UTC) before using this format specifier.
U	Universal sortable date/time pattern	Represents a custom **DateTime** format string defined by the current **FullDateTimePattern** property. This pattern is the same as the full date/long time (F) pattern. However, formatting operates on the Coordinated Universal Time (UTC) that is equivalent to the **DateTime** object being formatted.
Y or y	Year month pattern	Represents a custom **DateTime** format string defined by the current YearMonthPattern property. For example, the custom format string for the invariant culture is "yyyy MMMM".
Any other single character	(Unknown specifier)	An unknown specifier throws a runtime format exception.

Control Panel Settings

The settings in the **Regional and Language Options** item in Control Panel influence the result string produced by a formatting operation. Those settings are used to initialize the DateTimeFormatInfo object associated with the current thread culture, which provides values used to govern formatting. Computers using different settings will generate different result strings.

DateTimeFormatInfo Properties

Formatting is influenced by properties of the current **DateTimeFormatInfo** object, which is provided implicitly by the current thread culture or explicitly by the IFormatProvider parameter of the method that invokes formatting. Specify for the **IFormatProvider** parameter a CultureInfo object, which represents a culture, or a **DateTimeFormatInfo** object.

Many of the standard **DateTime** format specifiers are aliases for formatting patterns defined by properties of the current **DateTimeFormatInfo** object. Therefore, your application can change the result produced by some standard **DateTime** format specifiers by changing the corresponding **DateTimeFormatInfo** property.

Using Standard Format Strings

The following code fragment illustrates how to use the standard format strings with **DateTime** objects.

```
// This code example demonstrates the ToString(String) and
// ToString(String, IFormatProvider) methods for the DateTime
// type in conjunction with the standard date and time
// format specifiers.

using System;
using System.Globalization;
using System.Threading;

    function fn()
    {
    msgShortDate =  (d) Short date: . . . " . . .  ;                    "
    msgLongDate  =  (D) Long date: . . . . " . . .  ;                   "
    msgShortTime =  (t) Short time: . . . " . . .  ;                    "
    msgLongTime  =  (T) Long time: . . . . " . . .  ;                   "
    msgFullDateShortTime =
                        (f) Full date/short time: . .  ;
    msgFullDateLongTime =
                        (F) Full date/long time": . . .  ;
    msgGeneralDateShortTime =
                        (g) General date/short "time: .  ;
    msgGeneralDateLongTime =
                        (G) General date/long "time (default): \n  +
                            . . . . . " . . .  ;
    msgMonth    =    (M) Month: . . . . . " . . .  ;                    "
    msgRFC1123 =     (R) RFC1123: . . . . . " . . .  ;                  "
    msgSortable =    (s) Sortable: . . . . " . . .  ;                   "
    msgUniSortInvariant =
                        (u) Universal sortable "(invariant): \n  +
                            . . . . . " . . .  ;
    msgUniSort =     (U) Universal sortable:" . . .  ;                  "
    msgYear =        (Y) Year: . . . . . " . . .  ;                     "

    msg1 =  Use ToString(String) and the current thread culture.\n ;
    msg2 =  Use ToString(String, IFormatProvider) and a specified culture.\n ;
    msgCulture  =  Culture: ;              "        "
    msgThisDate =  This date and time: {0}"\n ;                    "

    thisDate  = DateTime.Now;
    utcDate  = thisDate.ToUniversalTime();

// Format the current date and time in various ways.
    print String.Format( Standard DateTime Format Specifiers:\n );
    print String.Format(msgThisDate, thisDate);
    print String.Format(msg1);

// Display the thread current culture, which is used to format the values.
    ci = Thread.CurrentThread.CurrentCulture;
    print String.Format( {0,-30}{1}\n , msgCulture, ci.DisplayName);

    print String.Format(msgShortDate           + thisDate.ToString( d ));
    print String.Format(msgLongDate            + thisDate.ToString( D ));
    print String.Format(msgShortTime           + thisDate.ToString( t ));
    print String.Format(msgLongTime            + thisDate.ToString( T ));
```

```
    print String.Format(msgFullDateShortTime    + thisDate.ToString( f ));
    print String.Format(msgFullDateLongTime     + thisDate.ToString( F ));
    print String.Format(msgGeneralDateShortTime + thisDate.ToString( g ));
    print String.Format(msgGeneralDateLongTime  + thisDate.ToString( G ));
    print String.Format(msgMonth                + thisDate.ToString( M ));
    print String.Format(msgRFC1123              +  utcDate.ToString( R ));
    print String.Format(msgSortable             + thisDate.ToString( s ));
    print String.Format(msgUniSortInvariant     +  utcDate.ToString( u ));
    print String.Format(msgUniSort              + thisDate.ToString( U ));
    print String.Format(msgYear                 + thisDate.ToString( Y ));
    print String.Format();

// Display the same values using a CultureInfo object. The CultureInfo class
// implements IFormatProvider.
    print String.Format(msg2);

// Display the culture used to format the values.
    ci = new CultureInfo( de-DE );              "     "
    print String.Format( {0,-30}{1}\n , msgCulture, ci.DisplayName);

    print String.Format(msgShortDate            + thisDate.ToString( d , ci));
    print String.Format(msgLongDate             + thisDate.ToString( D , ci));
    print String.Format(msgShortTime            + thisDate.ToString( t , ci));
    print String.Format(msgLongTime             + thisDate.ToString( T , ci));
    print String.Format(msgFullDateShortTime    + thisDate.ToString( f , ci));
    print String.Format(msgFullDateLongTime     + thisDate.ToString( F , ci));
    print String.Format(msgGeneralDateShortTime + thisDate.ToString( g , ci));
    print String.Format(msgGeneralDateLongTime  + thisDate.ToString( G , ci));
    print String.Format(msgMonth                + thisDate.ToString( M , ci));
    print String.Format(msgRFC1123              +  utcDate.ToString( R , ci));
    print String.Format(msgSortable             + thisDate.ToString( s , ci));
    print String.Format(msgUniSortInvariant     +  utcDate.ToString( u , ci));
    print String.Format(msgUniSort              + thisDate.ToString( U , ci));
    print String.Format(msgYear                 + thisDate.ToString( Y , ci));
    print String.Format();
    }
}
/A
```

This code example produces the following results:

```
Standard DateTime Format Specifiers:

This date and time: 1/9/2006 4:20:35 PM

Use ToString(String) and the current thread culture.

Culture:                      English (United States)

(d) Short date: . . . . . 4/17/2006
(D) Long date: . . . . . Monday, April 17, 2006
(t) Short time: . . . . . 2:38 PM
(T) Long time: . . . . . 2:38:09 PM
(f) Full date/short time: . . Monday, April 17, 2006 2:38 PM
(F) Full date/long time: . . . Monday, April 17, 2006 2:38:09 PM
(g) General date/short time: . 4/17/2006 2:38 PM
(G) General date/long time (default): . . 4/17/2006 2:38:09 PM
(M) Month: . . . . . April 17
(R) RFC1123: . . . . . Mon, 17 Apr 2006 21:38:09 GMT
(s) Sortable: . . . . . 2006-04-17T14:38:09
(u) Universal sortable (invariant): . . . 2006-04-17 21:38:09Z
(U) Universal sortable: . . . Monday, April 17, 2006 9:38:09 PM
(Y) Year: . . . . . April, 2006
(o) Roundtrip (local): . . . . 2006-04-17T14:38:09.9417500-07:00
```

```
(o) Roundtrip (UTC): . . . . . 2006-04-17T21:38:09.9417500Z
(o) Roundtrip (Unspecified): . 2000-03-20T13:02:03.0000000

Use ToString(String, IFormatProvider) and a specified culture.

Culture:                       German (Germany)

(d) Short date: . . . . . 17.04.2006
(D) Long date: . . . . . Montag, 17. April 2006
(t) Short time: . . . . . 14:38
(T) Long time: . . . . . 14:38:09
(f) Full date/short time: . . Montag, 17. April 2006 14:38
(F) Full date/long time: . . . Montag, 17. April 2006 14:38:09
(g) General date/short time: . 17.04.2006 14:38
(G) General date/long time (default): . . . . 17.04.2006 14:38:09
(M) Month: . . . . . 17 April
(R) RFC1123: . . . . . Mon, 17 Apr 2006 21:38:09 GMT
(s) Sortable: . . . . . 2006-04-17T14:38:09
(u) Universal sortable (invariant): . . 2006-04-17 21:38:09Z
(U) Universal sortable: . . . Montag, 17. April 2006 21:38:09
(Y) Year: . . . . . April 2006
(o) Roundtrip (local): . . . . 2006-04-17T14:38:09.9417500-07:00
(o) Roundtrip (UTC): . . . . . 2006-04-17T21:38:09.9417500Z
(o) Roundtrip (Unspecified): . 2000-03-20T13:02:03.0000000


A/
```

Standard DateTime Format Strings Output Examples

The following table illustrates the output created by applying some standard **DateTime** format strings to a particular date and time. Output was produced using the **ToString** method.

The Format string column indicates the format specifier, the Culture column indicates the culture associated with the current thread, and the Output column indicates the result of formatting.

The different culture values demonstrate the impact of changing the current culture. The culture can be changed by the settings in the **Regional and Language Options** item in Control Panel, or by passing your own DateTimeFormatInfo or CultureInfo class as the format provider. Note that changing the culture does not influence the output produced by the 'r' and 's' formats.

Short Date Pattern

Format string	Current culture	Output
d	en-US	4/10/2001
d	en-NZ	10/04/2001
d	de-DE	10.04.2001

Long Date Pattern

Format string	Current culture	Output
D	en-US	Tuesday, April 10, 2001

Long Time Pattern

Format string	Current culture	Output
T	en-US	3:51:24 PM
T	es-ES	15:51:24

Full Date/Time Pattern (Short Time)

Format string	Current culture	Output
f	en-US	Tuesday, April 10, 2001 3:51 PM
f	fr-FR	mardi 10 avril 2001 15:51

RFC1123 Pattern

Format string	Current culture	Output
r	en-US	Tue, 10 Apr 2001 15:51:24 GMT
r	zh-SG	Tue, 10 Apr 2001 15:51:24 GMT

Sortable Date/Time Pattern (ISO 8601)

Format string	Current culture	Output
s	en-US	2001-04-10T15:51:24
s	pt-BR	2001-04-10T15:51:24

Universal Sortable Date/Time Pattern

Format string	Current culture	Output
u	en-US	2001-04-10 15:51:24Z
u	sv-FI	2001-04-10 15:51:24Z

Month Day Pattern

Format string	Current culture	Output

m	en-US	April 10
m	ms-MY	10 April

Year Month Pattern

Format string	Current culture	Output
y	en-US	April, 2001
y	af-ZA	April 2001

An Invalid Pattern

Format string	Current culture	Output
L	en-UZ	Unrecognized format specifier; a format exception is thrown.

This is a more detailed description of the DateTime formatter.

Standard Numeric Format Strings

Standard numeric format strings are used to format common numeric types. A standard numeric format string takes the form **Axx**, where **A** is an alphabetic character called the format specifier, and **xx** is an optional integer called the precision specifier. The precision specifier ranges from 0 to 99 and affects the number of digits in the result. Any numeric format string that contains more than one alphabetic character, including white space, is interpreted as a custom numeric format string.

The following table describes the standard numeric format specifiers. For examples of the output produced by each format specifier, see Standard Numeric Format Strings Output Examples. For more information, see the notes that follow the table.

Format specifier	Name	Description
C or c	Currency	The number is converted to a string that represents a currency amount. The conversion is controlled by the currency format information of the current NumberFormatInfo (`□nfi`) object. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default currency precision given by the current NumberFormatInfo (`□nfi`) object.
D or d	Decimal	This format is supported only for integral types. The number is converted to a string of decimal digits (0-9), prefixed by a minus sign if the number is negative. The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier.
E or e	Scientific (exponential)	The number is converted to a string of the form "-d.ddd…E+ddd" or "-d.ddd…e+ddd", where each 'd' indicates a digit (0-9). The string starts with a minus sign if the number is negative. One digit always precedes the decimal point. The precision specifier indicates the desired number of digits after the decimal point. If the precision specifier is omitted, a default of six digits after the decimal point is used. The case of the format specifier indicates whether to prefix the exponent with an 'E' or an 'e'. The exponent always consists of a plus or minus sign and a minimum of three digits. The exponent is padded with zeros to meet this minimum, if required.
F or f	Fixed-point	The number is converted to a string of the form "-ddd.ddd…" where each 'd' indicates a digit (0-9). The string starts with a minus sign if the number is negative. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision given by the current NumberFormatInfo (`□nfi`) object.
G or g	General	The number is converted to the most compact of either fixed-point or scientific notation, depending on the type of the number and whether a precision specifier is present. If the precision specifier is omitted or zero, the type of the number determines the default precision, as indicated by the following list. • **Byte** or **SByte**: 3 • **Int16** or **UInt16**: 5

		• **Int32** or **UInt32**: 10 • **Int64** or **UInt64**: 19 • **Single**: 7 • **Double**: 15 • **Decimal**: 29 Fixed-point notation is used if the exponent that would result from expressing the number in scientific notation is greater than -5 and less than the precision specifier; otherwise, scientific notation is used. The result contains a decimal point if required and trailing zeroes are omitted. If the precision specifier is present and the number of significant digits in the result exceeds the specified precision, then the excess trailing digits are removed by rounding. The exception to the preceding rule is if the number is a **Decimal** and the precision specifier is omitted. In that case, fixed-point notation is always used and trailing zeroes are preserved. If scientific notation is used, the exponent in the result is prefixed with 'E' if the format specifier is 'G', or 'e' if the format specifier is 'g'.
N or n	Number	The number is converted to a string of the form "-d,ddd,ddd.ddd…", where '-' indicates a negative number symbol if required, 'd' indicates a digit (0-9), ',' indicates a thousand separator between number groups, and '.' indicates a decimal point symbol. The actual negative number pattern, number group size, thousand separator, and decimal separator are specified by the current NumberFormatInfo object. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision given by the current NumberFormatInfo object.
P or p	Percent	The number is converted to a string that represents a percent as defined by the NumberFormatInfo.PercentNegativePattern property if the number is negative, or the NumberFormatInfo.PercentPositivePattern property if the number is positive. The converted number is multiplied by 100 in order to be presented as a percentage. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision given by the current NumberFormatInfo object.
R or r	Round-trip	This format is supported only for the Single and Double types. The round-trip specifier guarantees that a numeric value converted to a string will be parsed back into the same numeric value. When a numeric value is formatted using this specifier, it is first tested using the general format, with 15 spaces of precision for a **Double** and 7 spaces of precision for a **Single**. If the value is successfully parsed back to the same numeric value, it is formatted using the general format specifier. However, if the value is not successfully parsed back to the same numeric value, then the value is formatted using 17 digits of precision for a **Double** and 9 digits of precision for a **Single**. Although a precision specifier can present, it is ignored. Round trips are given precedence over precision when using this specifier.
X or x	Hexadecimal	This format is supported only for integral types. The number is converted to a string of hexadecimal digits. The case of the format specifier indicates whether to use uppercase or lowercase characters for the hexadecimal digits greater than 9. For example, use 'X' to produce "ABCDEF", and 'x'

		to produce "abcdef". The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier.
Any other single character	(Unknown specifier)	(An unknown specifier throws a runtime format exception.)

Notes

Control Panel Settings

The settings in the **Regional and Language Options** item in Control Panel influence the result string produced by a formatting operation. Those settings are used to initialize the NumberFormatInfo object associated with the current thread culture, and the current thread culture provides values used to govern formatting. Computers using different settings will generate different result strings.

NumberFormatInfo Properties

Formatting is influenced by properties of the current **NumberFormatInfo** object, which is provided implicitly by the current thread culture or explicitly by the IFormatProvider parameter of the method that invokes formatting. Specify a **NumberFormatInfo** or CultureInfo object for that parameter.

Integral and Floating-Point Numeric Types

Some descriptions of standard numeric format specifiers refer to integral or floating-point numeric types. The integral numeric types are Byte, SByte, Int16, Int32, Int64, UInt16, UInt32, and UInt64. The floating-point numeric types are Decimal, **Single**, and **Double**.

Floating-Point Infinities and NaN

Note that regardless of the format string, if the value of a **Single** or **Double** floating-point type is positive infinity, negative infinity, or Not a Number (NaN), the formatted string is the value of the respective PositiveInfinitySymbol, NegativeInfinitySymbol, or NaNSymbol property specified by the currently applicable **NumberFormatInfo** object.

Example

The following code example formats an integral and a floating-point numeric value using the thread current culture, a specified culture, and all the standard numeric format specifiers. This code example uses two particular numeric types, but would yield similar results for any of the numeric base types (**Byte**, **SByte**, **Int16**, **Int32**, **Int64**, **UInt16**, **UInt32**, **UInt64**, **Decimal**, **Single**, and **Double**).

This example provides an excellent example of discreetly formatting an individual scalar and accessing resource information about the formatting object. □fmt applies these formatting techniques to arrays as well as scalars.

```
// This code example demonstrates the ToString(String) and
// ToString(String, IFormatProvider) methods for integral and
// floating-point numbers, in conjunction with the standard
// numeric format specifiers.
// This code example uses the System.Int32 integral type and
// the System.Double floating-point type, but would yield
// similar results for any of the numeric types. The integral
// numeric types are System.Byte, SByte, Int16, Int32, Int64,
// UInt16, UInt32, and UInt64. The floating-point numeric types
// are Decimal, Single, and Double.

using System;
using System.Globalization;
using System.Threading;

    function fn()
```

```
    {
// Format a negative integer or floating-point number in various ways.
    integralVal = -12345;
    floatingVal = -1234.567d;

    msgCurrency =     (C) Currency: . . . . ." .  ;                    "
    msgDecimal  =     (D) Decimal: . . . . . ." .  ;                   "
    msgScientific =   (E) Scientific: . . . ." .  ;                    "
    msgFixedPoint =   (F) Fixed point: . . . ." .  ;                   "
    msgGeneral =      (G) General (default): ." .  ;                   "
    msgNumber =       (N) Number: . . . . . ." .  ;                    "
    msgPercent =      (P) Percent: . . . . . ." .  ;                   "
    msgRoundTrip =    (R) Round-trip: . . . ." .  ;                    "
    msgHexadecimal =  (X) Hexadecimal: . . . ." .  ;                   "

    msg1 =  Use ToString(String) and the current thread culture.\n ;
    msg2 =  Use ToString(String, IFormatProvider) and a specified culture.\n ;
    msgCulture     =  Culture: ;             "        "
    msgIntegralVal =  Integral value: ;      "              "
    msgFloatingVal =  Floating-point value: ;"                    "

    CultureInfo ci;
    print  Standard Numeric Format" Specifiers:\n ;                   "
// Display the values.
    print msg1;

// Display the thread current culture, which is used to format the //values.
    ci = Thread.CurrentThread.CurrentCulture;
    print String.Format( {0,-26}{1} , msgCulture", ci.Displa'yName);

// Display the integral and floating-point values.
    print String.Format( {0,-26}{1} , msgIntegra'lVal, integ'ralVal);
    print String.Format( {0,-26}{1} , msgFloatin'gVal, float'ingVal);
    print                       ""
// Use the format specifiers that are only for integral types.
    print ( Format specifiers only"for integral types: );                "
    print String.Format(msgDecimal     + integralVal.ToString( D ));
    print String.Format(msgHexadecimal + integralVal.ToString( X ));
    print   ;                   ""

// Use the format specifier that is only for the Single and Double
// floating-point types.
    print ( Format specifier only for the Single and Double types: );
    print String.Format(msgRoundTrip   + floatingVal.ToString( R ));
    print   ;                   ""

// Use the format specifiers that are for integral or floating-point //types.
    print String.Format( Format specifiers for i"ntegral or floating-point
types: );                  "
    print String.Format(msgCurrency    + floatingVal.ToString( C ));
    print String.Format(msgScientific  + floatingVal.ToString( E ));
    print String.Format(msgFixedPoint  + floatingVal.ToString( F ));
    print String.Format(msgGeneral     + floatingVal.ToString( G ));
    print String.Format(msgNumber      + floatingVal.ToString( N ));
    print String.Format(msgPercent     + floatingVal.ToString( P ));
    print   ;                   ""

// Display the same values using a CultureInfo object. The //CultureInfo
class
// implements IFormatProvider.
    print (msg2);

// Display the culture used to format the values.
// Create a European culture and change its currency symbol to  euro  //
```

```
because this particular code example uses a thread current UI // // culture
that cannot display the euro symbol ( ).
    ci = new CultureInfo( de-DE );              "     "
    ci.NumberFormat.CurrencySymbol =  euro ;                "    "
    print  String.Format( {0,-26}{1} , msgCulture", ci.DisplayName);

// Display the integral and floating-point values.
    print  String.Format( {0,-26}{1} , msgIntegral"Val, integralVal);
    print  String.Format( {0,-26}{1} , msgFloating"Val, floating"Val);
    print   ;                    ""

// Use the format specifiers that are only for integral types.
    print  ( Format specifiers only"for integral types: );                    "
    print  String.Format(msgDecimal+ integralVal.ToString( D , ci));
    print  String.Format(msgHexadecimal+integralVal.ToString( X , ci));
    print   ;                    ""

// Use the format specifier that is only for the Single and Double
// floating-point types.
    print  String.Format( Format specifier only for the Single and Double
types: );                    "
    print  String.Format(msgRoundTrip+floatingVal.ToString( R , ci));
    print   ;                    ""

// Use the format specifiers that are for integral or floating-point types.
    print  String.Format( Format specifiers for i"ntegral or floating-point
types: );                    "
    print  String.Format(msgCurrency+floatingVal.ToString( C , ci));
    print  String.Format(msgScientific+floatingVal.ToString( E , ci));
    print  String.Format(msgFixedPoint+floatingVal.ToString( F , ci));
    print  String.Format(msgGeneral + floatingVal.ToString( G , ci));
    print  String.Format(msgNumber + floatingVal.ToString( N , ci));
    print  String.Format(msgPercent + floatingVal.ToString( P , ci));
    print   ;                    ""
    }

/A
```

This code example produces the following results:

Standard Numeric Format Specifiers:

Use ToString(String) and the current thread culture.

```
Culture:                  English (United States)
Integral value:           -12345
Floating-point value:     -1234.567

Format specifiers only for integral types:
(D) Decimal:. . . . . -12345
(X) Hexadecimal:. . . . . FFFFCFC7

Format specifier only for the Single and Double types:
(R) Round-trip: . . . . . -1234.567

Format specifiers for integral or floating-point types:
(C) Currency: . . . . . ($1,234.57)
(E) Scientific: . . . . . -1.234567E+003
(F) Fixed point:. . . . . -1234.57
(G) General (default):. . -1234.567
(N) Number: . . . . . -1,234.57
```

```
(P) Percent: . . . . . -123,456.70 %

Use ToString(String, IFormatProvider) and a specified culture.

Culture:                  German (Germany)
Integral value:           -12345
Floating-point value:     -1234.567

Format specifiers only for integral types:
(D) Decimal: . . . . . -12345
(X) Hexadecimal: . . . . . FFFFCFC7

Format specifier only for the Single and Double types:
(R) Round-trip: . . . . . -1234,567

Format specifiers for integral or floating-point types:
(C) Currency: . . . . . -1.234,57 euro
(E) Scientific: . . . . . -1,234567E+003
(F) Fixed point: . . . . . -1234,57
(G) General (default): . . -1234,567
(N) Number: . . . . . -1.234,57
(P) Percent: . . . . . -123.456,70 %

A/
```

This shows the formatting specifiers for the DateTime object.

□fmt

This documentation describes the supported feature set of the legacy □fmt system function.

□format in Visual APL provides support for all of the .Net formatting modifiers across arrays.

□fmt feature set:

□fmt - legacy formatter which returns character matrices with fixed width columns

The following elements of the legacy □fmt have been implemented for compatibility purposes.

Syntax:

```
res = 'fstring' □fmt data
```

'fstring' : character vector containing one or more editing phrases.

data : an array

Editing phrases:

rmAw Character

rmEw.s Exponential

rmFw.d Fixed point

rmG<pattern> Pattern

rmIw Integer

d = Decimal positions

s = Significant digits

w = Field width

<pattern> = Example

Positioning and text phrases:

r = Repetition (optional)

m = Modifiers (optional)

Modifiers:

B Blank if zero (F,I)

C Comma insertion (F,I)

L Left justify (F,I)

M<text> Negative left decoration (F,G,I)

N<text> Negative right decoration (F,G,I)

P<text> Non-negative left decoration (F,G,I)

Q<text> Non-negative right decoration (F,G,I)

Z Zero fill (F,I)

Valid delimiters for text in decorations are:

<text> ⊂text⊃ ¨text¨

□text□ ⍰text⍰ /text/

[ ] Index

Many classes have indexers.

Array indexer:

When used inside of an indexer bracket block [ ] the **;** axis separator identifies the values for each axis.

```
      a = 1 2 3
      a[1]
2
      a = 3 3⍴⍳9
      a[1 2;1 2]
 4 5
 7 8
```

It is not required to use the axis separator to index an array, for instance:

```
      b = (1 2) (1 2)
      a[b]
 4 5
 7 8
            b = 1 2
      a[b]
5
```

Providing a single value will index the array as though it were a vector.

```
       a[1]
1
```

You can select all values in an axis by using null:

```
      b = (1 2) (1 2) null
      a[b]
 12 13 14
 15 16 17

 21 22 23
 24 25 26
```

This makes it possible to index an array without having to be concerned about the syntax of the number of semi colons.

Generic Type Indexer

Indexers also occur on Generic Types. To create a Generic Type you need to first use:

```
      using System.Collections.Generic
      a = Dictionary[string, int]()
```

This will create an instance of the generic Dictionary type which accepts only string as the key, and int as the value.

```
      a.Add( test , 10)
      a.Add(100, 20)
bad args for method
      a.Count
1
```

It is not possible to use a key other than string with this Dictionary.

Method Selection Indexer

The signature of a method includes not only the name of the method, but also the types and number of arguments of the method.

To pre-select a particular method, indexing is available. As an example, an instance of string has a method named IndexOf which has 9 overloads. To select a specific overload:

```
      a =  test                    "    "
      a.IndexOf[string, int]( es ,1)                "  "
1
      a.IndexOf( es ,1)
1
```

In the vast majority of cases using the method indexer is not needed, but in some cases it can be quite beneficial. However, if the goal is to let the system select the best method for the dynamic values being used as arguments, then do not use the indexer.

← Assignment By Value and = Assign By Reference

The left assign arrow assigns data by value. This means that a copy of the data is made if possible. If it is not possible to make a copy of the data, a reference assignment is made.

Because this provides control over when assignment by value and assignment by reference will be made, discretion should be used when choosing to do assignment by value as copying all the data is considerably more expensive than assignment by reference. In general, there are relatively few occasions when assignment by value is required, which is one of the reasons it does not exist in other .Net languages.

For objects that are composed of ValueTypes, the copy is always made. However, for example, if an array contains an instance of a Form, then the Form is assigned by reference as creating another copy of the Form could have unintended consequences.

The = symbol is used for assign by reference, which matches the assignment behavior of other .Net languages. The ≈ symbol is used for comparison, or the double == symbol.

Example:

```
      a = ⍳10
      b←a
      a[3] = 100
      a
0 1 2 100 4 5 6 7 8 9
      b
0 1 2 3 4 5 6 7 8 9
      a = ⍳10
      b = a
      a[3] = 100
      a
0 1 2 100 4 5 6 7 8 9
      b
0 1 2 100 4 5 6 7 8 9
```

Simple assignment:

```
      a ← 10
      a b c ← 10 20 30
```

Assigns one value to each variable

```
      a b c ← ⊂10 20 30
```

Assigns the nested array 10 20 30 into each variable.

It is also possible to assign nested arrays by nesting shape.

```
      a (b c) d = 10 20 30
This makes a: 10, b: 20, c: 20 and d: 30

      a (b c) d = 10 (20 30) 40
In this case a: 10, b: 20, c: 30, d: 40
      x = 10 (20 (30 40)) 50
      a (b c) d = x
a: 10, b: 20, c: 30 40, d: 50
```

These assignment rules also apply when using for loops.

Matrix assignment:

```
      a←3 3⍴⍳9
      a
 0 1 2
 3 4 5
 6 7 8
      a[1 2;1 2]←2 2⍴10
      a
 0  1  2
 3 10 10
 6 10 10
```

Inline assignment works as follows:

```
      a ← 1+b ← 10+4
      a
15
      b
14
```

Selective assignment is also supported and is based on the original definition of selective assignment created by Jim Brown in his paper "Understanding Selective Assignment", 1989

"The notion of selective assignment is simple. If you can write an expression which selects some items at any depth in an array, then writing that same expression on the left of an assignment arrow requests replacement of the selected items."

This makes it possible to include user defined functions, the each operator, assign to more than one variable, etc.

For example:

```
      a = 1 2 3 4 5
      (1⊃a) = 10
      a = (1 2 3) (4 5 6)
      (1⊃¨a)=10
      (test a)=10
      (1 test a)=10
      a = 1 2 3 4
      b = 10 20 30 40
      ((1⊃a) (1⊃b)) = 100
      a
1 100 3 4
      b
10 100 30 40
      etc.
```

Execute

Compiles and runs a string which can be an expression or statement.

```
      ♦  1+1                    "   "
2
      ♦  a = 10+3               "       "
13
      a
13
```

It is also possible to manage the executes use of local and global variables. Execute can only create global variables, local variables can not be created with execute.

```
function fn(a) {
      b = 10
      c = 20
      ♦ c = a+b               "       "
      print a
}
      fn(10)
30
```

When it is desired to pass only a subset of local variables to the execute domain:

```
function fn(a) {
    b = 10
    c = 10
    d = 20
    // only local variables a and b passed to the execute
    ♦ c = a+b  in (a,b)     "       "
    print c
      // the value of c is not changed
      // a b and c are passed
    ♦ c = a+b  in (a,b,c)   "       "
    print c
}
```

It is also possible to manage the global variables passed and have new variables created added to the provided Dictionary. In this example we are not passing any local variables to execute, but we could include those as well. Functions can also be localized to the excute by placing them in the dictionary. In the case the function associated with fn in the dictionary does not exist in the class or session, but only in the dictionary.

```
      d = Dictionary[object, object]()
      d.Add( var1 , 20)           "    "
      d.Add( var2 , 30)           "    "
      d.Add( x , 40)              " "
      ♦ q = var1+var2+x  in (),"d             "
false
      d.Count
5
      d[ q ]                  " "
90
      ♦ q = var1+var2+x  in (),"d             "
false
      d[ var1 ] = 200         "    "
      ♦ q = var1+var2+x  in (),"d             "
false
```

```
      d[ q ]                    " "
270
      d.Add( fn ,    r←(a,b){r←a+b})"   "      f
      ⍝ q=fn(var1,var2)  in (),"d                "
false
      d[ q ]                    " "
50
```

All of the variables used and created by the execute come from the Dictionary object. The Dictionary object inherits from IDictionary and you can create a class which inherits from IDictionary which can respond in any desired way to the execution of the code and the creation and modification of variables. For instance, you could have an event fire when a new variable is created or a value is changed, or any other action you might find useful.

This provides detailed control of the execute, and provides the ability to scope function and variables to a particular execute.

Zilde

Empty numeric constant object.

This is displayed when the result of an expression evaluated in the session contains empty numeric data

⍕ Pattern format, Format

Simple formatter that provides simple width control and converts objects to their string representation. Relies on ⎕nfi

```
      ⍕2 3⍴⍳6
0 1 2
3 4 5
      (2 3⍴⍳6).ToString()
0 1 2
3 4 5
```

The ToString method in most cases is equivalent.

```
      1 0 4 1 6 2 ⍕ 2 3⍴⍳6
0 1.0  2.00
3 4.0  5.00
```

Notice that the width of each column was controlled by the left argument. The left argument is composed of value pairs, width and number of decimals.

Using a negative value for number of decimals formats objects in Exponential.

```
      10 ¯5 ⍕10 20 30 999.4
  1.0000E1  2.0000E1  3.0000E1  9.9940E2
```

The Share File System

The ShareFileSystem in Visual APL is a next generation component file system.

Not only does the ShareFileSystem support the legacy syntax common to share file systems, but it extends share file systems with virtual directories. This means you can place more than one share file in a single physical file.

To use the Share File System in your application, you will need to add a reference to the Visual APL Share/Native File System assembly. Here is an example of "referencing" and "using" the assembly by its strong name:

```
refbyname APLNext.APL.LegacyOps
using APLNext.Legacy.ShareFileSystem
```

The more Share Files that are placed in a virtual directory the better the space management becomes.

Additionally, because the ShareFileSystem uses the ISerializer .Net methodology for the IO of nested or object data types, shared and native files can read and write not only simple APL variables, but nested APL variables which even include Hashtables, Dictionaries, etc.

You can also write out the Hashtables or Dictionaries without including them in an APL variable.

Any class that inherits from ISerializable can be written to the share or native files and retrieved with the instance being automatically recreated.

□falloc

Pre-allocates a specific contiguous block in a component file as a single component.

```
      □falloc 12,1000
7
      ρ□fread 12,7
1000
```

Using this in conjunction with the index read (□firead) and index replace (□fireplace) you can easily manipulate text documents in a component.

It is also possible to retrieve the location of a component. This permits using other tools, such as □nread to access the data in a component. For instance, you could store a document in a component file, use □fcnloc to retrieve the starting point and then read the data using other tools:

```
      □fcnloc 12,7
54288
```

This is particularly useful to include images, documents and other data in a component file in a single virtual directory which needs to be accessed by other programs and tools.

□fappend

Appends a serializable object to a component file tied to the associated tie number. The append returns the component number into which the data was placed.

```
cn =  hello how are you   □fd'ppend 10          "
cn = (3 3⍴⍳10) □fappend 10
```

□fcatenate

One of the new features of these component files is the ability to manipulate component data in place. This means that it is not necessary to read in a component and catenate data, then write the component back out. Since catenate is one of the most expensive operations, this can be very useful. Only homogenous intrinsic data types can be manipulated in place. For instance a vector of integers, doubles, chars, etc. can be modified. However, nested arrays can not.

Example:

```
      (⍳5) □fappend 12
4
      □fread 12,4
0 1 2 3 4
      10 11 12 □fcatenate 12,4
      □fread 12,4
0 1 2 3 4 10 11 12
```

□fdrop

□fdrop removes components from the beginning or end of a Share File.

Syntax:

```
□fdrop tn dropCount
```

tn: The tie number of the file to drop components from.

dropCount: An integer specifying the number of components to drop from the file.

Remarks:

□fdrop will remove the specified number of components from either the beginning or end of the specified share file.

If the *dropCount* is a positive number, that number of components will be removed from the beginning of the Share File. If the *dropCount* is a negative integer, then that number of components will be removed from the end of the Share File.

Legacy Considerations

□fdrop duplicates the syntax of the legacy □fdrop, but has one difference, when you drop components from the front of a file, the components that remain are renumbered from 1 instead of retaining their original numbers. Since the Share File System is structured to give data back to the virtual pool, artificially numbering component offsets after a drop would have introduced many unwanted exceptions to the Share File System.

Example:

```
// drop 5 components from the beginning of the share
// file at tie number 1.
□fdrop 1 5
```

□ferase

Removes a specified component file from a virtual directory. This does not delete a physical file. The tie number must be the number associated with the file name to be erased.

```
       filename   □ferase 10  "         "
```

□fcreate

Has two primary uses:

1. Create a component file and associated virtual directory of the same name.
For instance:

```
"some file name.extension" □fcreate 10
```

Or

```
tn = □fcreate "some file name.extension"
```

Which returns the next available tie number.

Both of these create a file in the current directory. You could also specify the entire path:

```
tn = □fcreate @"c:\mydir\subdir\some file name.extension"
```

This use primarily exists for legacy system support. All of the above examples create a vitual directory with the same name as the fileid specified. This example further illustrates the point:

```
@ c: \test\myfile □fcreate 1 "
```

In the above example, a virtual directory is created with the same name as the fileid, "myfile", in the "c:\test" directory, and then creates a share file in that virtual directory with the same name.

Advantages over legacy file systems

One of the primary advantages of the Share File System is that not only can you place more than one share file in the virtual directories, but the share file system recovers data as it becomes available, thus avoiding the explosion of size common in some legacy share file systems.

Note
The use of the @ symbol to indicate a raw string, this obviates the need to use the \ as an escape character, as "c:\\mydir\\subdir\\some file name.extension"

2. When used with a library number, it creates a component file in the virtual directory associated with the library number.

```
"100 some file name.extension" □fcreate 10
```

Or

```
tn = □fcreate 10 some file name.extension" "
// the system chose the tie number, as none was specified
```

Or

```
tn = "100 some file name.extension" □fcreate 0
// the system chose the tie number, as a 0 was specified.
```

□firead

This provides the ability to read a subset of an intrinsic array using index read. This reduces the need to read large amounts of data into memory for the purpose of indexing only a subset. Used in conjunction with □ fireplace and □fcatenate it makes the management of discrete data within a component file very simple.

```
      (⍳20) □fappend 12
6

      □firead 12,6,10,3
10 11 12

       how are you today  □fappend 12          "
7
      □firead 12,7,4,3
are
```

□fireplace

The data in a component file can also be replaced in place using **index replace**: This obviates the need to read the data into memory, make the change, and then rewrite the data to disk. In this case the data is replaced on disk explicitly without reading the entire component into memory.

```
      (⍳20) □fappend 12
6
      100 200 300 □fireplace 12,6,10,3
      □fread 12,6
0 1 2 3 4 5 6 7 8 9 100 200 300 13 14 15 16 17 18 19
```

Catenating and modifying integers in place is extremely useful when updating pointers, such as are used as references. This significantly reduces the time and space required to maintain systems which require reading and modifying large arrays of integers, doubles, characters, etc.

In the event more data is provided than allocated for by the arguments, then only the first n elements of the data is used in the replacement:

```
      85 86 87 88 □fireplace 12,6,10,3

      □fread 12,6
0 1 2 3 4 5 6 7 8 9 85 86 87 13 14 15 16 17 18 19
```

□fnames

Returns a string array of strings. This is useful for manipulation with .Net classes such as generic List.

```
      a = □fnames
      a.GetType()
System.String[]
```

fnums

Returns an integer array of tie numbers indicating all of the files currently associated with a tie number.

□fread

Reads a component from a component file. The syntax for this is:

```
a = □fread tn cn
```

Any arbitrary serializable data can be returned from a component. The data will be deserialized and the original object will be returned.

□freplace

Replaces the data in an existing component with an arbitrary serializable object.

```
a = 1 20 30 40.5
a □freplace tn cn
```

□fsize

Returns a five element integer vector.

```
      □fsize
1 10  0  0  0
```

The first element is the starting component, the second element is the next component which will be used. The last component in use is this element less one.

□libdup

□libdup duplicates an entire virtual directory based on the associated library number, releasing any unused space from the virtual pool of the library.

Syntax:

```
newLibNo dupPath  □libdup libNo        "
```

newLibNo: The library number to which *dupPath* will be associated.

dupPath: The file path at which to create the newly duplicated library.

tn: The library number for the Share File library to duplicate.

Remarks:

The □libdup system function creates a copy of the specified library.

This newly created copy of the file library contains all components and data which were present in the source library.

The only difference between the source and newly created libraries, is that the newly created library has had all unused space released from the virtual pool of the Share File.

This process decreases the physical file size of the library, since all unused space in the library has been released back to the operating system.

The inclusion of the □libdup system function is primarily for completeness in the Share File System, as the Share File System by design reclaims space as necessary from the virtual pool.

Example:

```
 @ 2 c:\test\testnew  □libdup 101          "

Where 101 is the library number for the existing virtual directory. This
will duplicate all of the files in the 101 virtual directory and place
them in c:\test\testnew which is associated with the library number 2.
```

□fdup

Visual APL includes □fdup for legacy support.

Syntax:

```
filePath □fstream tn
```

filePath: The full file path of the tied share file.

tn: The tie number of the file to dup.

Remarks:

□fdup duplicates a single file. This will only duplicate share files whose name matches the virtual directory in which they reside, and the virtual directory contains only the file being duplicated.

Example:

```
c:\myfiles\filename  □fdup 3                "
```

□fremove

□fremove removes the specified component from a Share File, and renumbers the remaining components.

Syntax:

```
□fremove tn compNumber
```

tn: The tie number of the file to drop the component from.

compNumber: An integer specifying the component number to remove from the file.

Remarks:

□fremove removes a single component from a Share File, returning the space used by the removed component to the virtual pool.

Example:

```
// drop component 10 from the share
// file at tie number 2.
□fremove 2 10
```

□fstream

Returns the underlying .Net FileStream object for the associated tie number. This allows the use of all features provided by the FileStream object, while still maintaining compatibility with the Share File system.

```
      fs = □fstream 3

      fs.CanRead
true
      fs.CanWrite
true
```

□fstie

Ties an existing file and associates the file with either a given tie number or the next available tie number.

```
       c: \myfiles\filename  □fstie 10            "
      or
      tn = □fstie  c: \myfiles\filename    "                  "
      or
      // if a tie number of 0 is specified, the system assigns the next
available tie number.
       c: \myfiles\filename  □fstie 0             "
```

It is also possible to access component files within a virtual directory created either with □libd or □fcreate by using the associated library number for a virtual directory.

```
       101 filename  □fstie 10"            "

      tn = □fstie  10 filename           "           "

       101 filename  □fstie 0"            "
```

In this way many component files can reside in a virtual directory, or single physical file.

funtie

Removes the tie number associated with the existing component file.

□lib

To manage your files in their virtual directory, you have □fnums and □fnames as well as □lib and □libs:

```
            □lib 10

'my2file'   'myfile'   'another'
```

Which returns an array of file names found in the virtual directory.

To remove a file from a virtual directory, use:

```
       another   □ferase 12    "        "

      □lib 10
'my2file'   'myfile'
```

To untie a file use □funtie.

□libd

Since component files reside in a single physical file, to create the physical file or virtual directory you use □libd, for instance:

```
      □libd  10 c: \\tmysf          "             "
true
```

Notice that the directory path has two backslashes, as the \ is the escape character. You could have also placed an @ symbol at the beginning for raw text, for instance:

```
      □libd @ 10 c: \tmysf          "             "
true
```

Which obviates the need for the double backslash.

Once the virtual directory has been created, you can use it just like you normally use a library.

For instance:

```
      □fcreate  10 myfile             "          "
1
      □fsize 1
1 1 0 0 0

      10 □fappend 1
1
```

The component file can also be tied or created by specifying the tie number:

```
       10 another  □fcreate 12'          "
12
      □fsize 12
1 1 0 0 0
```

The tie number can be changed at any time by simply retieing:

```
       10 another  □ftie 10   "          "
10
```

As with all component files, you can store disparate data types in the components and retrieve them, as well as replace component data:

```
       test  □fappend 12      "     "
1

      10 11 12 □fappend 12
2

      □fread 12,2
10 11 12

       test  (10 11 12)  mores"tuff" □fappend 12 "          "
3

      □fread 12,3
 test  10 11 12 morestuff
```

```
      (3 3   9) □freplace 12,2
      □fread 12,2
 0 1 2
 3 4 5
 6 7 8
```

One of the new features of these component files is the ability to manipulate component data in place. This means that it is not necessary to read in a component and catenate data, then write the component back out. Since catenate is one of the most expensive operations, this can be very useful. Only homogenous intrinsic data types can be manipulated in place. For instance a vector integers, doubles, chars, etc. can be modified. However, nested arrays can not.

Example:

```
      ( 5) □fappend 12
4
      □fread 12,4
0 1 2 3 4
      10 11 12 □fcatenate 12,4
      □fread 12,4
0 1 2 3 4 10 11 12
```

This file system also uses blocks to minimize file size explosion as component sizes grow.

It is also possible to manage character data:

```
      hello how are   □fappend 12          "
5
       you?  □fcatenate 12,5  "    "

      □fread 12,5
hello how are you?
```

It is also possible to read a subset of an intrinsic array using index read:

```
      ( 20) □fappend 12
6

      □firead 12,6,10,3
10 11 12
```

The data can also be replaced in place using index replace:

```
      100 200 300 □fireplace 12,6,10,3
      □fread 12,6
0 1 2 3 4 5 6 7 8 9 100 200 300 13 14 15 16 17 18 19
```

Catenating and modifying integers in place is extremely useful when updating pointers, such as are used as references. This significantly reduces the time and space required to maintain systems which require reading and modifying large arrays of integers, double, characters, etc.

In the event more data is provided than allocated for by the arguments, then only the first n elements of the data is used in the replacement:

```
      85 86 87 88 □fireplace 12,6,10,3

      □fread 12,6
0 1 2 3 4 5 6 7 8 9 85 86 87 13 14 15 16 17 18 19
```

It is also possible to allocate a contiguous block of space as a single component:

```
      □falloc 12,1000
7
```

```
      □fread 12,7
1000
```

Using this in conjunction with the index read and replace you can easily manipulate text documents in a component.

It is also possible to retrieve the location of a component. This permits using other tools, such as □nread to access the data in a component. For instance, you could store a document in a component file, use □ fcnloc to retrieve the starting point and then read the data using other tools:

```
     □fcnloc 12,7
54288
```

This is particularly useful to include images, documents and other data in a component file in a single virtual directory which needs to be accessed by other programs and tools.

□libdcws

It is also possible to control access to the virtual directory, this is done with □libdrw for setting read only or read/write access, and □libdcws for checking write status. Use 0 to set read only and 1 for read/write.

```
      □libdcws 10
1

      □libdrw 10,0
0

      □libdcws 10
0

      □libdrw 10,1
1

      □libdcws 10
1
```

□libdrw

It is also possible to control access to the virtual directory, this is done with □libdrw for setting read only or read/write access, and □libdcws for checking write status. Use 0 to set read only and 1 for read/write.

```
      □libdcws 10
1

      □libdrw 10,0
0

      □libdcws 10
0

      □libdrw 10,1
1

      □libdcws 10
1
```

□libs

This displays a matrix of all virtual directories and their associated library numbers.

```
      □libs
         2 C: \mydir\test.mf
         3 C: \mydir\test
```

Visual APL provides a large set of primitives which include both functions and operators. These are represented by symbols that specify which operations to perform in an expression. Visual APL predefines the usual arithmetic, data manipulation and logical functions and operators, as well as a variety of others as shown in the following table. In addition, many operators can be overloaded by the user, thus changing their meaning when applied to a user-defined type. There are two facilities provided to achieve this overloading, one is the using of a class with the appropriate attributes in place and the second is the overloading of .Net common operators, which can be overloaded in C# using the operator keyword. With the using keyword it is also possible to add functions and operators.

The primitive functions and operators provide support for all of the intrinsic .Net datatypes. As such, long, short, float, double, etc will be referred to as numeric. As there are a large number of intrinsic datatypes as well as Complex, IntN, BitArray, etc. not all types are included in the default array operator set. The default types are Int32, Double, and Char. However, scalar operations on all datatypes will work for the .Net base operator set.

In Visual APL, a function or operator is a term or a symbol that takes one or more expressions, called operands, as input and returns a value. Operators that take one operand, such as the increment operator (++), are called monadic or unary operators. Operators that take two operands, such as arithmetic operators (+,-,*,/) are called dyadic or binary operators. One operator.

The following Visual APL statement contains a single monadic operator, and a single operand. The increment operator, **++**, modifies the value of the operand `y`.:

Visual APL

```
y++;
```

The following Visual APL statement contains two dyadic operators, each with two operands. The assignment operator, **=**, has the integer `y`, and the expression `2 + 3` as operands. The expression `2 + 3` itself contains the addition operator, and uses the integer values `2` and `3` as operands:

Visual APL

```
y = 2 + 3;
```

An operand can be a valid expression of any size, composed of any number of other operations. Operators in an expression are evaluated in a specific order, that is right to left. The following table divides the operators into categories based on the type of operation they perform.

Primary	x.y, f(x), a[x], x++, x--, new, typeof
Monadic (scalar and array)	+, -, !, ~, (T)x, ⍴, ×, ÷, ⍳, ∈, ⌊, ⌈, ↑, ↓
Dyadic (scalar and array)	(,ravel) , !, ?, ⍋, ⍒, ⌽, ⊖, ⊂, ⊃, ⍱, ⍲, ⍉
Arithmetic --- Multiplicative (scalar and array)	**×, ÷, |,** ⍟**, *,** ○

Arithmetic --- Multiplicative (scalar)	%
Arithmetic --- Additive (scalar and array)	+, -
Shift (scalar)	<<, >>
Relational (scalar and array)	<, >, <=, >=, ≤ ≥
Type testing (scalar)	is, as
Equality (scalar and array)	==, ≈ ≡ ≠ ≅, ≣
Equality (scalar)	!=
Logical (scalar and array)	⋀, ⋁, ⍲, ⍱, ~
Logical (scalar)	&, ^, |
Data Analysis (scalar and array)	⍳, ∈, ⌈, ⌊, ⊥, ⊤, !, ?, ⍋, ⍒, ⍷, ⍸, ⍭
Data Manipulation (scalar and array)	⍴, ↑ ↓ (,catenate), ⍪, ⊂, ⊃, ⊖, ⌽, ⍉
Operator Functions (scalar and array)	/, \, [], (. dot), ¨, ⍀, ⌿, ∘ ⍨
Conditional (Boolean)	&&, ||, then/else
Assignment	=, ← +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, ...

Dyadic Operators are evaluated from right to left, Monadic (unary) operators are evaluated from left to right.

```
Visual APL

num1 = 5;

num1++;

print num1
```

However, the output of the following example code is undefined:

```
Visual APL

num2 = 5;

num2 = num2++;  //not recommended

print num2
```

Therefore, the latter example is not recommended. Parentheses can be used to surround an expression and force that expression to be evaluated before any others. For example, $2 \times 3 + 4$ would normally become 14. This is because dyadic operators evaluate from right to left . Writing the expression as $(2 \times 3) + 4$ results in 10, because it indicates to the Visual APL compiler that the multiplication operator (×) must be evaluated before the addition operator (**+**).

The Add function can act as either a monadic or dyadic primitive.

```
result ← expr1 + expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The dyadic + functions are predefined for numeric and string types. For numeric types, + computes the sum of its two operands. When one or both operands are of type string, + concatenates the string representations of the operands.

User-defined types can overload the dyadic + functions.

Example

```
function fn() {
    □ ← 10 + 10
    □ ← 10.5 + 10.5
    □ ←  hello   +  world    "      "   "     "
    □ ← 2j + 4j
    □ ← 2.0 +   2                   "  "
}

    fn()
20
21
hello world
 6j
2 2

(only scalar Complex numbers are supported in this version)
```

The ∧ function can act as either a monadic or a dyadic primitive.

```
result ← expr1 ∧ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dyadic ∧ functions are predefined for the integral types. For integral types and arrays of integrals, ∧ computes the logical AND of its operands.

0 is always treated as false, all other values including 1 are treated as true.

Example

```
function fn() {
    □ ← 1 0 1 0 ∧ 1 0 1 0
    □ ← 0 1 0 1 ∧ 0 0 0 0
    □ ← 1 ∧ 1
    □ ← 1 ∧ 0
    □ ← 0 ∧ 0
    □ ← 1 2 3 4 ∧ 4 3 2 1
}

      fn()
1 0 1 0
0 0 0 0
1
0
0
1 1 1 1
```

Specifies that *operatorexpr1* should apply its functionality across the dimension(s) specified in *axisexpr*

```
result ← expr1 operatorexpr1[ axisexpr ] expr2
result ← operatorexpr1[ axisexpr ] expr2
```

Where:

result
An expression.

expr1
An expression.

operatorexpr1
An operator expression.

axisexpr
An axis expression.

expr2
An expression.

Remarks

The Axis operator provides a mechanism for applying the functionality of *operatorexpr1* to *expr1* and *expr2* across the dimension or dimensions specified by axisexpr.

axisexpr is a numeric vector.

Example

```
function fn() {
    □ ←  apply a function acrd'ss first axis                    "
    □ ← Φ[0]3 3⍴⍳9
    □ ←  apply a function acrd'ss second axis                    "
    □ ← Φ[1]3 3⍴⍳9
    □ ←  apply a function witH' reduction across first axis                    "
    □ ← +/[0]3 3⍴⍳9
    □ ←  apply a function witH' reduction across second axis                    "
    □ ← +/[1]3 3⍴⍳9
    □ ←  apply a function witH' scan across first axis                    "
    □ ← +\[0]3 3⍴⍳9
    □ ←  apply a function witH' scan across second axis                    "
    □ ← +\[1]3 3⍴⍳9
    □ ←  apply a dyadic functi"on across first axis                    "
    □ ← 1Φ[0]3 3⍴⍳9
    □ ←  apply a dyadic functi"on across second axis                    "
    □ ← 1Φ[1]3 3⍴⍳9
    □ ←  apply a dyadic functi"on with reduction across first axis                    "
    □ ← 2+/[0]4 4⍴⍳16
    □ ←  apply a dyadic functi"on with reduction across second axis                    "
    □ ← 2+/[1]4 3⍴⍳16
    □ ←  apply a dyadic functi"on with scan across first axis                    "
    □ ← 2+\[0]4 4⍴⍳16
    □ ←  apply a dyadic functi"on with scan across second axis                    "
    □ ← 2+\[1]4 4⍴⍳16
}

      fn()
apply a function across first axis
 6 7 8
 3 4 5
 0 1 2
apply a function across second axis
 2 1 0
 5 4 3
 8 7 6
apply a function with reduction across first axis
9 12 15
apply a function with reduction across second axis
3 12 21
apply a function with scan across first axis
 0  1  2
 3  5  7
 9 12 15
apply a function with scan across second axis
 0  1  3
 3  7 12
 6 13 21
```

```
apply a dyadic function across first axis
 3 4 5
 6 7 8
 0 1 2
apply a dyadic function across second axis
 1 2 0
 4 5 3
 7 8 6
apply a dyadic function with reduction across first axis
  4  6  8 10
 12 14 16 18
 20 22 24 26
apply a dyadic function with reduction across second axis
  1  3
  7  9
 13 15
 19 21
apply a dyadic function with scan across first axis
  4  6  8 10
 12 14 16 18
 20 22 24 26
apply a dyadic function with scan across second axis
  1  3  5
  9 11 13
 17 19 21
 25 27 29
```

Determines the number of groups of objects in the population represented by expr2 based on group size defined by expr1.

```
result ← expr1   expr2        !
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Binomial function supports positive arrays of numbers, and negative arrays of numbers.

Example

```
function fn() {
    □ ← 2   10          !
    □ ← 2 3 4   10 11 12     !
    □ ← 2   ¯10          !
    □ ← 2 3 4   ¯10 20 ¯30   !
}

      fn()
45
45 165 495
55
55 1140 40920
```

Square brackets ([]) are used for arrays, indexers, attributes, and dynamic generic selection.

```
type[]
array[ indexexpr ]
generictype[ typeexpr ]
```

Where:

type
A type.

array
An array.

indexexpr
An index expression.

generictype
A generic type.

typeexpr
A type expression.

Remarks

An array type is defined as a type followed by brackets:

```
int[] a = new int[10]
or dynamic
a = 0 0 0 0 0 0 0 0 0 0
```

To access an element of an array, the indices of the desired elements are enclosed in brackets after the expression:

Dependent state: □IO

```
      a = 10 20 30
      a[0]
10
      a[0 1]
10 20
```

The array indexing operator cannot be overloaded; however, types can define indexers, properties that take one or more parameters. Indexer parameters are enclosed in square brackets, just like array indices, but indexer parameters can be declared to be of any type (unlike array indices, which must be integral).

Example

```
function fn() {
    a = 1 2 3 4 5
    □ ← a[0 1 2]
    a = 3 3⍴⍳9
    □ ← a[0 1; 0 1]
    □ ← a[(0 1) (0 2)]
    a = Hashtable()
    a[ test ] = ⍳10          "     "
    □ ← a[ test ]                "     "
    a = Dictionary[string, int]()
    a.Add( one , 10)             "    "
    □←a[ one ]                 "    "
}

      fn()
1
 0 1
 3 4
1 2
0 1 2 3 4 5 6 7 8 9
10
```

The Catenate function can act as either a monadic or dyadic primitive.

```
result ← expr1 , expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Catenates *expr1* with *expr2* along the last axis, unless another axis is provided.

Scalar expressions are expanded to conform with the non scalar expression.

Array expressions which differ by a rank of 1 are expanded to be conformable with the higher rank expression. Arrays must match in primary dimensions.

Example

```
function fn() {
    a = 1 2 3
    b = 4 5 6
    □ ← a, b
    a =  test                  "    "
    b =  more                  "    "
    □ ← a, b
    a = 3 3⍴⍳9
    b = 3 4⍴⍳12
    □ ← a, b
    a = 10.4
    □ ← a, b
    a =  test  10              "    "
    b =  more  20              "    "
    □ ← a, b
}
      fn()
1 2 3 4 5 6
testmore
 0 1 2 0 1  2  3
 3 4 5 4 5  6  7
 6 7 8 8 9 10 11
 10.4 0 1  2  3
 10.4 4 5  6  7
 10.4 8 9 10 11
 test 10  more 20
```

Returns the smallest whole number greater than or equal to the specified number.

```
return ← ⌈ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

Dependent state: ⎕CT

The Floor function returns the smallest whole number greater than or equal to a. If a is equal to NaN, NegativeInfinity, or PositiveInfinity, that value is returned.

The behavior of this function follows IEEE Standard 754, section 4. This kind of rounding is sometimes called rounding toward positive infinity.

Example

```
function fn() {
    ⎕ ← ⌈ 100.5
    ⎕ ← ⌈ 100.7
    ⎕ ← ⌈ 100.2
    ⎕ ← ⌈ 100.1 200.1 300.1
    ⎕ ← ⌈ 3 3⍴10.2
}
      fn()
101
101
101
101 201 301
 11 11 11
 11 11 11
 11 11 11
```

The Replicate function can act as either a monadic or dyadic primitive.

```
result ← expr1 / expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

expr1 must be a vector equal in length to the last dimension of expr2. If another axis is specified, then the length of expr1 must match the length of the specified dimension of expr2.

For values of 0 in expr1, elements in expr2 are removed. For positive integral elements in expr1, elements in expr2 are replicated integral times.

Example

```
function fn() {
    □ ← 0 1 0 1 / 1 2 3 4
    □ ← 0 1 0 1 / 4 4⍴⍳16
    □ ← 1 2 3 4 / 1 2 3 4
}
      fn()
2 4
  1  3
  5  7
  9 11
 13 15
1 2 2 3 3 3 4 4 4 4
```

The Depth function can act as either a monadic or dyadic primitive.

```
result ←≡expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Depth function determines the deepest level of nesting present in expr1.

Example

```
function fn() {
    a = 1
    ⎕←≡
    a = 1 2
    ⎕←≡
    a =  test  2              "    "
    ⎕←≡
    a = cc 2 3
    ⎕←≡
}
      fn()
0
1
2
3
```

The Disclose function can act as either a monadic or dyadic primitive.

```
result ← ⊃ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Disclose function builds result from the elements of expr1.

If expr1 has only one (1) element, result is simply the contents of that first element. This simple case of Disclose is also known as un-nest, since it removes one (1) level of nesting from the data of expr1.

If expr1 contains two (2) or more elements, each element of expr1 is conformed such that every element of expr1 has the same rank and shape, and these elements are then structured into the result. The result is structured by concatenating together the conformed elements of expr1, and reshaping the result to be the shape of expr1 concatenated with the determined conformed shape applied to the elements of expr1.

The Disclose function is the inverse of the Enclose function.

Example

```
function fn() {
    a = 1
    □ ← ⊃ a
    □ ← ⍴ ⊃ a
    a = 1 2 3
    □ ← ⊃ a
    □ ← ⍴ ⊃ a
    a = ⊂⊂1 2 3
    □ ← ⊃ a
    □ ← ⍴ ⊃ a
    a = (1 2 3) 2 3
    □ ← ⊃ a
    □ ← ⍴ ⊃ a
    a = (3 3⍴1 2 3) 2 3
    □ ← ⊃ a
    □ ← ⍴ ⊃ a
}
      fn()
1

1 2 3
3
 1 2 3

 1 2 3
 2 0 0
 3 0 0
3 3
 1 2 3
 1 2 3
 1 2 3

 2 0 0
 0 0 0
 0 0 0

 3 0 0
 0 0 0
 0 0 0
3 3 3
```

Divide ÷

The ÷ function can act as either a monadic or a dyadic primitive.

```
result ← expr1 ÷ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The division operator (÷) divides its first operand by its second. All numeric types have predefined division operators.

Dependent state: ⎕DBZ, ⎕DBZV

The ⎕DBZ state variable provides control over the way in which divide addresses division by zero.

The default value is 0 to match .Net languages, however, you can set ⎕DBZ to the following:

```
⎕dbz:
 0 : 1÷0 = 0
     0÷0 = 0
 1 : 1÷0 = DOMAIN ERROR
     0÷0 = 1
 2 : 1÷0 = DOMAIN ERROR
     0÷0 = DOMAIN ERROR
 3 : 1÷0 = NaN or ⎕dbzv
     0÷0 = NaN or ⎕dbzv
 4 : 1÷0 = +-Infinity
     0÷0 = NaN
```

You can set ⎕DBZV to any object, and it will be returned when ⎕dbz is set to 3.

User-defined types can contain cross language overloads to the ÷ operator.

Example

```
function fn() {
    ⎕ ← 10 20 ÷
    ⎕ ← 20 10 ÷
    ⎕ ← 10 20 ÷ 20 10
    ⎕ ← 10.1 20.2 ÷ 10 20
    ⎕ ← 10 20 ÷ 10.1 20.1
    ⎕ ← (3 3⍴⍳9) ÷ 10
}

      fn()
0.5
2
0.5 2
1.01 1.01
0.9900990099 0.9950248756
   0 0.1 0.2
 0.3 0.4 0.5
 0.6 0.7 0.8
```

The Drop function can act as either a monadic or dyadic primitive.

```
result ← expr1 ↓ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Drop function removes data from dimensions of expr2, according to the amounts specified in expr1.

The length of expr1 should match the rank of expr2, and each element of expr1 specifies the amount of data to drop from the respective dimension of expr2.

The elements of expr1 can be either negative, positive, or 0. If an element of expr1 is positive, that length is dropped from the related dimension of expr2. If an element of expr1 is negative, that length is dropped from opposite end of the related dimension of expr2. If an element of expr1 is 0, no data is dropped from the related dimension of expr2.

Example

```
function fn() {
    □ ← 1 ↓ 10 11 12
    □ ← 3 ↓ 10 11 12 13 14 15
    □ ← ¯1 ↓ 10 11 12
    □ ← ¯3 ↓ 10 11 12 13 14 15
    □ ← 2 2 ↓ 3 3⍴⍳9
    □ ← ¯2 ¯2 ↓ 3 3⍴⍳9
}
      fn()
11 12
13 14 15
10 11
10 11 12
 8
 0
```

Performs the specified operator expression across each element of expr1 and expr2.

```
result ← expr1 operator¨ expr2
```

Where:

result
An expression.

expr1
An expression.

operator
An operator expression.

expr2
An expression.

Remarks

The Each operator is a specialized short hand construct simulating a single for loop across the elements of *expr2*.

The Each data iterator performs the specified operator expression between each element of expr1 and expr2. If expr1 or expr2 is a scalar, that expression is considered to be the same rank and shape of the higher rank expression.

Example

```
function fn() {
    □ ← 2 ⍴ ¨ 1 2 3
    □ ← (⊂2 2) ⍴ ¨ 1 2 3
    □ ← (⊂2 2) ⍴ ¨ (1 2) (⍳4)
    □ ← (⊂1 2 3) + ¨ (1 2 3) (10 20 30)
    □ ← (⊂⊂1 2 3) + ¨¨ (1 2 3) (10 20 30)
    □ ← 3 ⍳ ¨ (1 2 3) (4 5 6)
    □ ← (⊂2 3) ⍳ ¨ (2 3) (4 5) (5 6)
    □ ← ⍳ ¨ 1 2 3
    □ ← ⍴ ¨ (1 2) (3 4 5) (3 3⍴⍳9)
}
      fn()
 1 1  2 2  3 3
  1 1   2 2   3 3
  1 1   2 2   3 3
  1 2   0 1
  1 2   2 3
 2 4 6  11 22 33
  2 3 4  3 4 5  4 5 6   11 12 13  21 22 23  31 32 33
 1 1 0  1 1 1
 0 1  2 2  2 2
 0  0 1  0 1 2
 2  3  3 3
```

The Enclose function can act as either a monadic or dyadic primitive.

```
result ←⊂expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Enclose function creates *result* by nesting *expr1* once.

The only exception to this enclosure rule is if expr1 is a native .Net type scalar, such as Int32, Double, or Char. If expr1 is a .Net native type scalar, the data is not enclosed, and *result* is exactly equal to *expr1.*

The Enclose function is the inverse of the Disclose function.

Example

```
function fn() {
    a = ⊂
    ⎕← shape of enclosed scalar                    "
    ⎕←⍴
    a = ⊂ 2 3
    ⎕← shape of enclosed vector                    "
    ⎕←⍴
    a = ⊂1 2 3) (5 6 7)
    ⎕← shape of enclosed vector of vectors                  "
    ⎕←⍴
    a = ⊂(1 2 3) (4 5 6)
    ⎕← shape of enclose of each vector                   "
    ⎕←⍴
    ⎕← shape of each enclosed vector                   "
    ⎕←⍴a
    a = ⊂⊂ 2 3
    ⎕← shape of the original vector using each                    "
    ⎕←⍴¨a
}
      fn()
shape of enclosed scalar

shape of enclosed vector

shape of enclosed vector of vectors

shape of enclose of each vector
2
shape of each enclosed vector

shape of the original vector using each
  3
```

The Enlist function can act as either a monadic or dyadic primitive.

```
result ← ∈ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Enlist function produces a flattened version of *expr1*. *result* contains all data which was present in *expr1* and its sub elements, with all nesting, shape, and rank removed, so that *result* is a simple vector.

Example

```
function fn() {
    a = ∈1
    □ ← a
    □ ← ⍴a
    a = ∈1 2 3
    □ ← a
    □ ← ⍴a
    a = ∈3 3⍴⍳9
    □ ← a
    □ ← ⍴a
    a = ∈(1 2 3) (4 5 6)
    □ ← a
    □ ← ⍴a
    a = ∈(ccc1 2 3) (cc test )              "    "
    □ ← a
    □ ← ⍴a
}
      fn()
1
1
1 2 3
3
0 1 2 3 4 5 6 7 8
9
1 2 3 4 5 6
6
 1 2 3 test
7
```

The Approximately Equal function can act as either a monadic or dyadic primitive.

```
result ← expr1 ≈ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

`Dependent state: ⎕CT`

The Approximately Equal function returns a 1 if *expr1* is equal to *expr2*, or if *expr2* is within ⎕CT of *expr1*. Otherwise, the return is 0.

Example

```
function fn() {
    ⍝ ← 10 ≈ 12
    ⍝ ← 10 ≈ 9 10 11
    ⍝ ← 10 ≈ 5+3 3⍴⍳9
    ⍝ ← 1 2 3 ≈ 1 2 3
    ⍝ ← 1 2 3 ≈ 1+1 2 3
    ⍝ ← 1 2 3 ≈ 1.1 2.1 2.1
    ⍝ ← (3 3⍴10.1) ≈ 3 3⍴10 11
}
      fn()
0
0 1 0
 0 0 0
 0 0 1
 0 0 0
1 1 1
0 0 0
0 0 0
 0 0 0
 0 0 0
 0 0 0
```

Executes the code supplied by expr1

```
result ← ⍎ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Execute expression dynamically executes the code returned by *expr1*. *expr1* can return either a string, a dynamic variable (IVariable), or a compiled code object (obtainable through the compile method). If *expr1* evaluates to a string, then the code is parsed, compiled, and then executed. If *expr1* is a compiled code object, no parsing and compilation is required, and the code object is executed immediately.

Note: Language features which effect the code flow of a function do not effect the function which initiated the dynamic execution. Examples of these kinds of statements include yield, return, break, continue, branching, and conditional branching. Such statements can be used within the respective constructs to which they apply, such as a yield statement within a function defined in the same dynamic execution.

Advanced Dynamic Execution Features:

Dynamic execution allows you to override the module dictionaries used within the context of the dynamic execution. Using this feature, you can specify either or both of the local variable and global variable dictionaries, which enables the dynamic execution of code within contexts other than the context of the function which called the dynamic execution. You can even create entirely new contexts under program control just for the purpose of dynamically executing code.

The following example calls dynamic execute and specifies that only "a" and "b" are to be used in the local dictionary of the execution:

```
      a = 10
      b = 20 30 40
      c = ⍎ a+b  in (a,b)        "    "
      c
30 40 50
```

Depending on where an execute statement is programmed in your code, you will have access to either or both of the global dictionaries *ws* and *wsi*. The field *ws* contains all static data which exists in the current context of where you reference *ws*, and *wsi* contains all instance data for the context it which it is referenced.

In functions which are defined with the *static* access modifier, only the *ws* field will be accessible, because by definition no instance data can be referenced from a *static* method. In an instance method, or any method which does not exist in a static class or has the *static* modifier applied to its definition, you also have access to the global field *wsi*.

By default, when you run a dynamic execution and do not specify the global context in which it will run, the *wsi* (or *ws* for static methods) is passed as the default global dictionary.

Dynamically defining contexts:

You can dynamically create a global context under program control, which can be used in place of the default *ws* or *wsi* global fields.

Here is an example of creating a module dictionary which contains a single element "alist". Once the dictionary is created and initialized, the dictionary is then passed to execute as the global dictionary:

Note: Any object which inherits from IDictionary can be used as a global dictionary.

```
      using System
      using System.Collections
      using System.Collections.Generic
      gd = Dictionary[object, object]()
      a = ArrayList()
      a.Add(10)
0
      a.Add( test )              "    "
```

```
1
      a.Count
2
      gd.Add( alist ,a)            "     "
      ≛  alist.Add('more')  in' (),gd           "
false
      a.Count
3
```

As you can see above, the variable "alist" does not exist in the context in which the execute is run, and only exists as an entry in the newly created Dictionary object which was passed to the execute statement. Using this methodology, you can dynamically create any arbitrary context in which to run your dynamic execution.

Dynamic Evaluation:

All code which is processed by dynamic execute is fully compiled to the lowest possible level in .Net, which allows the code to run as fast as any code compiled at runtime. In some cases, the code statement to be run by execute may be small enough that the extra time required to compile the code would be unnecessary, and in these cases it may be optimal to interpret the code directly.

To directly interpret a code snippet, use the eval statement. Here is the above example for execute, modified to instead use the eval method:

```
      using System
      using System.Collections
      using System.Collections.Generic
      gd = Dictionary[object, object]()
      a = ArrayList()
      a.Add(10)
0
      a.Add( test )              "    "
1
      a.Count
2
      gd.Add( alist ,a)            "     "
      eval( alist.Add('more') , null, gd)          "
false
      a.Count
3
```

The performance gain of directly interpreting code is only found when evaluating small and simple snippets of code. While fully supported, snippets which include statements such as *for* or *while* loops would not be normally appropriate, because the iteration process re-evaluates each line of code as it is run in the for loop, and is therefore not as highly optimized as direct compilation.

Example

```
      a = 10
      b = 20 30 40
      ≛ a+b                  "   "
30 40 50
      c = ≛ a+b                "   "
      c
30 40 50
      c = ≛ a+b  in (a,b)        "   "
      c
30 40 50
      using System.Collections.Generic
      gd = Dictionary[object, object]()
      x
name 'x' is not defined
      c = ≛ x = a+b  in (a,b),gd"       "
      x
name 'x' is not defined
      gd[ x ]                " "
30 40 50
      using System.Collections
      h = Hashtable()
      gd[ newhash ] = h        "       "
      h.Count
0
```

```
      c = * newhash.Add(\ one\ ,100.9)  in (),gd    "       "
      gd[ newhash ].Count      "       "
1
      h.Count
1
      h[ one ]                 "   "
100.9
      e = compile( a = b+c ,ws)         "       "
      b = 10
      c = 100
      *e in (b,c)
110
      a
110
      *e
110
```

The Expand function can act as either a monadic or dyadic primitive.

```
result ← expr1 \ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The length of the *result* is determined by the length of *expr1.*

expr1 is a numeric vector, where at every non zero (0) element the next element of *expr2* will be inserted into the result. Where a zero (0) occurs in *expr1*, the fill data element for *expr2* is inserted into the *result* instead.

Example

```
function fn() {
    □ ← 1 0 1 0 1 \ 1 2 3
    □ ← 1 0 1 0 1 \ 3 3⍴⍳9
    □ ← 1 0 1 \  test  (1 2 3)        "    "
    □ ← 1 0 0 1 1 \ 3 3⍴⍳9
}
      fn()
1 0 2 0 3
 0 0 1 0 2
 3 0 4 0 5
 6 0 7 0 8
 test        1 2 3
 0 0 0 1 2
 3 0 0 4 5
 6 0 0 7 8
```

The Exponential function can act as either a monadic or dyadic primitive.

```
result ← ⋆ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Exponential function expands the Math.Exp method to work with numeric arrays.

Math.Exp returns the number **e** raised to the power *expr1*. If *expr1* equals *NaN* or *PositiveInfinity*, that value is returned. If *expr1* equals *NegativeInfinity*, 0 is returned.

Example

```
function fn() {
    □ ← ⋆0
    □ ← ⋆1
    □ ← ⋆2
    □ ← ⋆3
}

      fn()
1
2.718281828
7.389056099
20.08553692
```

The Factorial function can act as either a monadic or dyadic primitive.

```
result ←   expr1        !
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Factorial function determines the mathematical factorial of *expr1*. For non integral *expr1*, the standard mathematical procedure of determining the factorial result through the *Gamma* function is applied.

Example

```
function fn() {
    □ ←  1              !
    □ ←  2              !
    □ ←  3              !
    □ ←  4              !
    □ ←  1 2 3 4        !
}
      fn()
1
2
6
24
1 2 6 24
```

The Find function can act as either a monadic or dyadic primitive.

```
result ← expr1   expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Find function returns an integer array of the same shape and rank as *expr2*, with a one (1) wherever the array *expr1* was found in *expr2*. *expr1* and *expr2* can be arrays of any shape, rank, and depth.

Dependent state: CT

Example

```
function fn() {
    □ ← 1 2 3 ⍷ 1 2 3 4 1 2 3
    □ ← 0 1 2 ⍷ 3 3⍴⍳9
    □ ← what ⍷ morewhatofwhat "   "              "
    □ ← hey ⍷ 4 3⍴ heyyouheyyou"       "            "
}
      fn()
1 0 0 0 1 0 0
 1 0 0
 0 0 0
 0 0 0
0 0 0 0 1 0 0 0 0 0 1 0 0 0
 1 0 0
 0 0 0
 1 0 0
 0 0 0
```

The First function can act as either a monadic or dyadic primitive.

```
result ← ↑ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The First function returns the first element of *expr1,* disclosing the element if it is enclosed.

The First function is a short hand for accessing the first element of an array.

Example

```
function fn() {
    □ ← ↑ 1 2 3
    □ ← ↑ 3 3⍴⍳9
    □ ← ↑ 3 3 3⍴⍳27
    □ ← ↑  test                "    "
    □ ← ↑ 4 4⍴ test                 "    "
}
      fn()
1
0
0
t
t
```

The Floor function can act as either a monadic or dyadic primitive.

```
result ← ⌊ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

```
Dependent state:  CT
```

The Floor function returns the largest whole number less than or equal to *expr1*. If *expr1* is equal to *NaN*, *NegativeInfinity*, or *PositiveInfinity*, then that value is returned.

The behavior of this function follows IEEE Standard 754, section 4. This kind of rounding is sometimes called rounding toward negative infinity.

Note: The Floor function uses CT when determining if *expr1* is already equal to an integral value. If *expr1* is within CT of the next greater whole number, than the Floor function does not apply. Instead, Floor assumes that if *expr1* cannot be rounded to the next lesser whole number, than it must match the next greatest whole number, and the next greatest whole number is returned. This guarantees that only integers will return from the Floor function.

Example

```
function fn() {
    □ ← ⌊ 1.1
    □ ← ⌊ 1.5
    □ ← ⌊ 1.8
    □ ← ⌊ 1.1 1.5 1.8
    □ ← ⌊ 3 3⍴10.1 11.1 12.1
}

      fn()
1
1
1
1 1 1
 10 11 12
 10 11 12
 10 11 12
```

The Format function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍕ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dependent state: NFI, PP

The Format function Creates

Simple formatter that provides simple width control and converts objects to their string representation. Relies on ⎕nfi

```
⍕2 3⍴⍳6
 0 1 2
 3 4 5
      (2 3⍴⍳6).ToString()
 0 1 2
 3 4 5
```

The ToString method in most cases is equivalent.

```
      1 0 4 1 6 2 ⍕ 2 3⍴⍳6
0 1.0  2.00
3 4.0  5.00
```

Notice that the width of each column was controlled by the left argument. The left argument is composed of value pairs, width and number of decimals.
Using a negative value for number of decimals formats objects in Exponential.

```
      10 ¯5 ⍕10 20 30 999.4
  1.0000E1  2.0000E1  3.0000E1  9.9940E2
```

Example

```
function fn() {
    ⎕ ← ⍕1 2 3
    ⎕ ← 3 ⍕ 1.2 2.3 3.4
    ⎕ ← 7 2 ⍕ 1.2 2.3 3.4
    ⎕ ← 7 ¯2 ⍕ 1.2 2.3 3.4
    ⎕ ← 7 2 ⍕ 3 3⍴1.2 2.3 3.4
    ⎕ ← 1 0 6 2 7 3 ⍕ 2 3⍴1 2 3
}
      fn()
1 2 3
1.200 2.300 3.400
   1.20   2.30   3.40
  1.2E0  2.3E0  3.4E0
   1.20   2.30   3.40
   1.20   2.30   3.40
   1.20   2.30   3.40
1  2.00  3.000
1  2.00  3.000
```

Produces a vector of numbers, which is the representation of expr2 with radix specifications expr1.

```
result ← expr1 ⊤ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

From Base 10 (Encode) is the inverse function of To Base 10 (Decode)

Example

```
function fn() {
    □ ← 10 10 10 10 ⊤ 1776
    □ ←  Convert 3622 minutes "to 2 days, 12 hours, 22 minutes                    "
    □ ← 0 24 60 ⊤ 3622
    □ ←  Convert 10 to 8 bits "                    "
    □ ← 2 2 2 2 2 2 2 2 ⊤ 10
}
      fn()
1 7 7 6
Convert 3622 minutes to 2 days, 12 hours, 22 minutes
2 12 22
Convert 10 to 8 bits
0 0 0 0 1 0 1 0
```

The Grade Down function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍒ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state: ⎕IO
```

The Grade Down function returns an integer array of indices which specify the sorted order of *expr2,* **in descending order***,* according to either the order of *expr1* if it is supplied, or the IComparable interface implemented by the argument data in *expr2*.

The Grade functions extend the Microsoft Array.Sort method to work with arrays of all rank and depth.

The Microsoft Array.Sort method performs a highly optimized, unstable Q-Sort on the elements of vectors to be sorted, using the IComparable interface implemented by each element of the array being sorted for determining if one value is greater than another.

The Grade functions extend Array.Sort to function on arrays in general, and also stabilize the result so that elements which are considered equal appear in the result in the same order that they appeared in *expr2*. Also, if *expr2* is all of a single type, only one comparitor is utilized, further optimizing the sorting process.

If *expr1* is supplied, a custom comparitor is created which sorts the elements of *expr2* according to the order of their appearance in *expr1*. If an element of *expr2* does not exist in *expr1*, that element is considered to have the least importance in the sorting process, and will appear after all other elements in the result which did exist in *expr1*. Of course, all elements of the result which do not appear in *expr1* are stabilized as the sort progresses, and appear in the order in which they occurred in *expr2*.

Note that the result might vary depending on the current CultureInfo.

Note: The IComparable interface defines a generalized comparison method that a value type or class implements to create a type-specific comparison method. Visit the Microsoft web site to see examples of how to implement the IComparable interface on your Visual APL classes.

Example

```
function fn1() {
    a = 50 40 30 20 10
    ⎕ ← ⍒a
    ⎕ ← a[⍒a]
    a = 10 20 30 40 50
    ⎕ ← ⍒a
    ⎕ ← a[⍒a]
    a = 3 3⍴⍳9
    ⎕ ← ⍒a[;0]
    ⎕ ← a[⍒a[;0];]
    a =  abcde                "      "
    ⎕ ← ⍒a
    ⎕ ← a[⍒a]
    a = 3 3⍴ abcdefghi            "          "
    ⎕ ← ⍒a
    ⎕ ← a[⍒a;]
}
      fn1()
0 1 2 3 4
50 40 30 20 10
4 3 2 1 0
50 40 30 20 10
2 1 0
 6 7 8
 3 4 5
 0 1 2
4 3 2 1 0
edcba
```

```
2 1 0
ghi
def
abc
0 1 2
 1 2 3  2 3 4  3 4 5



function fn2() {
    a =  abcde                "     "
    c =  edcba                "     "
    □ ← c ⍒ a
    □ ← a[c⍒a]
    a = 1 2 3 4 5
    c = 5 4 3 2 1
    □ ← c ⍒ a
    □ ← a[c⍒a]
    a = 3 3⍴⍳9
    c = 9 8 7 6 5 4 3 2 1 0
    □ ← c ⍒ a
    □ ← a[c⍒a;]
    a = (1 2 3) ( test ) (3 4 5)       "    "
    c = (3 4 5) ( test ) (1 2 3)       "    "
    □ ← c ⍒ a
    □ ← a[c⍒a]
}

      fn2()
0 1 2 3 4
abcde
0 1 2 3 4
1 2 3 4 5
0 1 2
 0 1 2
 3 4 5
 6 7 8
0 1 2
 1 2 3  test  3 4 5
```

The Grade Up function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍋ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state: ⎕IO
```

The Grade Up function returns an integer array of indices which specify the sorted order of *expr2,* **in ascending order**, according to either the order of *expr1* if it is supplied, or the IComparable interface implemented by the argument data in *expr2*.

The Grade functions extend the Microsoft Array.Sort method to work with arrays of all rank and depth.

The Microsoft Array.Sort method performs a highly optimized, unstable Q-Sort on the elements of vectors to be sorted, using the IComparable interface implemented by each element of the array being sorted for determining if one value is greater than another.

The Grade functions extend Array.Sort to function on arrays in general, and also stabilize the result so that elements which are considered equal appear in the result in the same order that they appeared in *expr2*. Also, if *expr2* is all of a single type, only one comparitor is utilized, further optimizing the sorting process.

If *expr1* is supplied, a custom comparitor is created which sorts the elements of *expr2* according to the order of their appearance in *expr1*. If an element of *expr2* does not exist in *expr1*, that element is considered to have the least importance in the sorting process, and will appear after all other elements in the result which did exist in *expr1*. Of course, all elements of the result which do not appear in *expr1* are stabilized as the sort progresses, and appear in the order in which they occurred in *expr2*.

Note that the result might vary depending on the current CultureInfo.

Note: The IComparable interface defines a generalized comparison method that a value type or class implements to create a type-specific comparison method. Visit the Microsoft web site to see examples of how to implement the IComparable interface on your Visual APL classes.

Example

```
function fn1() {
    a = 50 40 30 20 10
    ⎕ ← ⍋a
    ⎕ ← a[⍋a]
    a = 10 20 30 40 50
    ⎕ ← ⍋a
    ⎕ ← a[⍋a]
    a = 3 3⍴⍳9
    ⎕ ← ⍋a[;0]
    ⎕ ← a[⍋a[;0];]
    a =  abcde               "     "
    ⎕ ← ⍋a
    ⎕ ← a[⍋a]
    a = 3 3⍴ abcdefghi           "         "
    ⎕ ← ⍋a
    ⎕ ← a[⍋a;]
}
      fn1()
4 3 2 1 0
10 20 30 40 50
0 1 2 3 4
10 20 30 40 50
0 1 2
 0 1 2
 3 4 5
 6 7 8
0 1 2 3 4
```

```
abcde
0 1 2
abc
def
ghi


function fn2() {
    a =  abcde                "     "
    c =  edcba                "     "
    □ ← c ⍋ a
    □ ← a[c⍋a]
    a = 1 2 3 4 5
    c = 5 4 3 2 1
    □ ← c ⍋ a
    □ ← a[c⍋a]
    a = 3 3⍴⍳9
    c = 9 8 7 6 5 4 3 2 1 0
    □ ← c ⍋ a
    □ ← a[c⍋a;]
    a = (1 2 3) ( test ) (3 4 5)       "    "
    c = (3 4 5) ( test ) (1 2 3)       "    "
    □ ← c ⍋ a
    □ ← a[c⍋a]
}
      fn2()
4 3 2 1 0
edcba
4 3 2 1 0
5 4 3 2 1
2 1 0
 6 7 8
 3 4 5
 0 1 2
2 1 0
 3 4 5  test  1 2 3
```

The Greater Than function can act as either a monadic or dyadic primitive.

```
result ← expr1 > expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dependent state: ⎕CT

The Greater Than function returns 1 if *expr1* is greater than *expr2*. Otherwise, the return is 0. All numeric and enumeration types define a "greater than" relational operator.

User-defined types can contain cross language overloads to the > operator.

Example

```
function fn() {
    ⎕ ← 10 > 12
    ⎕ ← 10 > 9 10 11
    ⎕ ← 10 > 5+3 3⍴⍳9
    ⎕ ← 1 2 3 > 1 2 3
    ⎕ ← 1 2 3 > 1+1 2 3
    ⎕ ← 1 2 3 > 1.1 2.1 2.1
    ⎕ ← (3 3⍴10.1) > 3 3⍴10 11
}
      fn()
0
1 0 0
 1 1 1
 1 1 0
 0 0 0
0 0 0
0 0 0
0 0 1
 1 0 1
 0 1 0
 1 0 1
```

The Greater Than or Equal function can act as either a monadic or dyadic primitive.

```
result ← expr1 ≥ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dependent state: ⎕CT

The Greater Than or Equal function returns 1 if *expr1* is greater than, or equal to, *expr2*. Otherwise, the return is 0. All numeric and enumeration types define a "greater than or equal" relational operator.

User-defined types can contain cross language overloads to the ≥ operator. If ≥ is overloaded, ≤ must also be overloaded.

Example

```
function fn() {
    ⎕ ← 10 ≥ 12
    ⎕ ← 10 ≥ 9 10 11
    ⎕ ← 10 ≥ 5+3 3⍴⍳9
    ⎕ ← 1 2 3 ≥ 1 2 3
    ⎕ ← 1 2 3 ≥ 1+1 2 3
    ⎕ ← 1 2 3 ≥ 1.1 2.1 2.1
    ⎕ ← (3 3⍴10.1) ≥ 3 3⍴10 11
}
      fn()
0
1 1 0
 1 1 1
 1 1 1
 0 0 0
1 1 1
0 0 0
0 0 1
 1 0 1
 0 1 0
 1 0 1
```

The IndexOf function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍳ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dependent state: ⎕IO, ⎕CT

The IndexOf function returns the index of the first occurrence of expr1 in expr2. If expr2 does not contain expr1, the returned index is one plus the number of elements in expr2.

The IndexOf function is similar in use to the IndexOf method found on many objects in .Net, with the exception of returning one plus the number of elements in the argument data, instead of a -1.

Example

```
function fn() {
    ⎕ ← 'hello world' ⍳ 'hello world' '' '' ''
    ⎕ ← 1 2 3 ⍳ 10 20 30 1 40 2 50 3 1 2 3
    ⎕ ← (⍳10) ⍳ 1 4 20
    ⎕ ← 0 1 2 3 ⍳ 3 3⍴⍳9
    ⎕ ← 1 ⍳ 3 2 1 3 2 1
}
      fn()
0 1 2 2 4 5 6 4 8 2 10
3 3 3 0 3 1 3 2 0 1 2
1 4 10
 0 1 2
 3 4 4
 4 4 4
1 1 0 1 1 0
```

The Inner Product function can act as either a monadic or dyadic primitive.

```
result ← expr1 operatorexpr1 . operatorexpr2 expr2
```

Where:

result
An expression.

expr1
An expression.

operatorexpr1
An operator expression.

operatorexpr2
An operator expression.

expr2
An expression.

Remarks

The Inner Product function is a specialized short hand construct for successively calling operators in a pre defined order.

The Inner Product function creates its result by first calling the function specified by *operatorexpr2* as though that function had been called dyadically with *expr1* and *expr2*, and then takes the result of that operation, and uses it as the right operand to the reduce version of *operatorexpr1*.

Example

```
function fn() {
    □ ← (3 3⍴⍳9) ∧.≈ 0 1 2
    □ ← (3 5⍴ hellowhatsupdoc ) ∧.≠  whats         "       "      "
    □ ← 1 2 3 +.× 1 2 3
    □ ← 10+.×(1 2 3) (4 5 6)
}
      fn()
1 0 0
0 1 0
14
 50 70 90
```

The Interval function can act as either a monadic or dyadic primitive.

```
result ← ⍳ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

`Dependent state:  ⎕IO`

The Interval function produces an integer vector from one (1) to *expr1*, or if `⎕IO` is zero (0), from zero (0) to (*expr1* - 1).

Example

```
function fn() {
    ⍞ ← ⍳10
    ⍞ ← 1+⍳10
    ⍞ ← 3+3×⍳10
}
      fn()
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
3 6 9 12 15 18 21 24 27 30
```

The Laminate function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍪ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Implicit argument: ⎕IO
```

Catenates *expr1* with *expr2* along the first axis, unless another axis is provided.

Scalar expressions are expanded to conform with the non scalar expression.

Array expressions which differ by a rank of 1 are expanded to be conformable with the higher rank expression. Arrays must match in primary dimensions.

Example

```
function fn() {
    a ← 1 2 3 ⍪ 1 2 3
    ⎕ ← a
    ⎕ ← ⍴a
    a ← 1 2 3 ⍪ 1 3⍴1 2 3
    ⎕ ← a
    ⎕ ← ⍴a
    a ← 1 2 3 ⍪ 3 3⍴⍳9
    ⎕ ← a
    ⎕ ← ⍴a
    a ← 1 ⍪ 1 3⍴1 2 3
    ⎕ ← a
    ⎕ ← ⍴a
    a ←  abc  ⍪ 2 3⍴ efghij   "   "        "       "
    ⎕ ← a
    ⎕ ← ⍴a
}
      fn()
1 2 3 1 2 3
6
 1 2 3
 1 2 3
2 3
 1 2 3
 0 1 2
 3 4 5
 6 7 8
4 3
 1 1 1
 1 2 3
2 3
abc
efg
hij
3 3
```

The Less Than function can act as either a monadic or dyadic primitive.

```
result ← expr1 < expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dependent state: ⎕CT

The Less Than function returns 1 if *expr1* is less than *expr2*. Otherwise, the return is 0. All numeric and enumeration types define a "less than" relational operator.

User-defined types can contain cross language overloads to the < operator.

Example

```
function fn() {
    ⍝ ← 10 < 12
    ⍝ ← 10 < 9 10 11
    ⍝ ← 10 < 5+3 3⍴⍳9
    ⍝ ← 1 2 3 < 1 2 3
    ⍝ ← 1 2 3 < 1+1 2 3
    ⍝ ← 1 2 3 < 1.1 2.1 2.1
    ⍝ ← (3 3⍴10.1) < 3 3⍴10 11
}
      fn()
1
0 0 1
 0 0 0
 0 0 0
 1 1 1
0 0 0
1 1 1
1 1 0
 0 1 0
 1 0 1
 0 1 0
```

The Less Than or Equal function can act as either a monadic or dyadic primitive.

```
result ← expr1 ≤ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state: ⎕CT
```

The Less Than or Equal function returns 1 if *expr1* is less than, or equal to, *expr2*. Otherwise, the return is 0. All numeric and enumeration types define a "less than or equal" relational operator.

User-defined types can contain cross language overloads to the ≤ operator. If ≤ is overloaded, ≥ must also be overloaded.

Example

```
function fn() {
    ⎕ ← 10 ≥ 12
    ⎕ ← 10 ≥ 9 10 11
    ⎕ ← 10 ≥ 5+3 3⍴⍳9
    ⎕ ← 1 2 3 ≥ 1 2 3
    ⎕ ← 1 2 3 ≥ 1+1 2 3
    ⎕ ← 1 2 3 ≥ 1.1 2.1 2.1
    ⎕ ← (3 3⍴10.1) ≥ 3 3⍴10 11
}
      fn()
0
1 1 0
 1 1 1
 1 1 1
 0 0 0
1 1 1
0 0 0
0 0 1
 1 0 1
 0 1 0
 1 0 1
```

The Logarithm function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍟ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Logarithm function expands the Math.Log methods to work with numeric arrays.

Example

```
function fn() {
    □ ← 2⍟4
    □ ← 2⍟8
    □ ← 2⍟16
    □ ← 2⍟32
    □ ← 10⍟100 1000 10000 100000
}
      fn()
2
3
4
5
2 3 4 5
```

The Magnitude function can act as either a monadic or dyadic primitive.

```
result ← ¦ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Magnitude function expands the Math.Abs method to work with numeric arrays.

Math.Abs returns the absolute value of a specified number.

Example

```
function fn() {
    □ ← | 10
    □ ← | ¯10
    □ ← | 10 ¯10 ¯3 2 ¯1
}

      fn()
10
10
10 10 3 2 1
```

The Match function can act as either a monadic or dyadic primitive.

```
result ← expr1 ≡ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

`Dependent state: ⎕CT`

The Match function returns a result of either 1 or 0. The result is 1 if *expr1* and *expr2* are identical in data, shape, rank, and depth, at all levels of nesting in *expr1* and *expr2*. Otherwise, the result is 0.

Example

```
function fn() {
    a = 1 2 3
    b = 1 2 3
    ⎕ ← a ≡ b
    a = 1
    b = 1
    ⎕ ← a ≡ b
    a =  test   what          "    " "    "
    b = 1 2 3  what                "    "
    ⎕ ← a ≡ b
    a =  more  1 2 3  of  4 5 "6   "       "  "
    b =  more  1 2 3  of  4 5 "6   "       "  "
    ⎕ ← a ≡ b
    a = 1 2 3
    b = 3 3⍴⍳9
    ⎕ ← a ≡ b
    a = 3 3⍴⍳9
    ⎕ ← a ≡ b
}
      fn()
1
1
0
1
0
1
```

The Matrix Divide function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⌹ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Solve, or least squares fit, a set of simultaneous equations where *expr1* is the vector of constants, and *expr2* is a matrix of coefficients.

Example

```
function fn() {
    // solve these linear equations using
    // matrix divide
    // 1x + 3y = 31
    // 4x + 4y = 68
    // 6x + 7y = 109
    □ ← 31 68 109 ⌹ 3 2⍴1 3 4 4 6 7
}
      fn()
10 7
```

The Matrix Inverse function can act as either a monadic or dyadic primitive.

```
result ← ⌹ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

Calculate the matrix inverse of *expr1*.

Example

```
function fn() {
    □ ← ⌹3
    □ ← ⌹3 2
    □ ← ⌹3 2 2
    □ ← ⌹3 2 2 3
    □ ← ⌹2 2⍴3 2 2 3
}
      fn()
0.3333333333
0.2307692308 0.1538461538
0.1764705882 0.1176470588 0.1176470588
0.1153846154 0.07692307692 0.07692307692 0.1153846154
  0.6 ¯0.4
 ¯0.4  0.6
```

The Maximum function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⌈ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Maximum function returns the larger of two specified numbers.

Example

```
function fn() {
    □ ← 4 ⌈ 20
    □ ← ¯3 ⌈ ¯6
    □ ← 10 ⌈ 11 5 13 6
    □ ← ¯5 ⌈ 10 ¯20 4 ¯2
    □ ← 5 4 5 4 ⌈ 6 3 4 6
}
      fn()
20
¯3
11 10 13 10
10 ¯5 4 ¯2
6 4 5 6
```

The Member function can act as either a monadic or dyadic primitive.

```
result ← expr1 ∈ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Implicit argument: ⎕CT
```

The Member function returns an integer 1 or 0, indicating whether *expr1* occurs within *expr2*. A result of 1 indicates that *expr1* occurs in *expr2*. Otherwise, the result is 0.

Example

```
function fn() {
    ⎕ ← 1 ∈ 1 2 3 1 2 3
    ⎕ ← 1 2 3 ∈ ⍳9
    ⎕ ← 30 40 1 2 ∈ ⍳9
    ⎕ ← (3 3⍴⍳9) ∈ ⍳5
    ⎕ ← "hello" "world" ∈ "what" "a" "world" "" "" "" "" "" ""
    ⎕ ← "testing" (1 2 3) ∈ (1 2 3) "testing" "" ""
}
      fn()
1
1 1 1
0 0 1 1
 1 1 1
 1 1 0
 0 0 0
0 1
1 1
```

The Minimum function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⌊ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Minimum function returns the larger of two specified numbers.

Example

```
function fn() {
    □ ← 4 ⌊ 20
    □ ← ¯3 ⌊ ¯6
    □ ← 10 ⌊ 11 5 13 6
    □ ← ¯5 ⌊ 10 ¯20 4 ¯2
    □ ← 5 4 5 4 ⌊ 6 3 4 6
}
      fn()
4
¯6
10 5 10 6
¯5 ¯20 ¯5 ¯5
5 3 4 4
```

The Multiply function can act as either a monadic or dyadic primitive.

```
result ← expr1 × expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The multiplication function (×) computes the product of its operands. All numeric types have predefined multiplication operators.

User-defined types can contain cross language overloads to the × operator.

Example

```
function fn() {
    □ ← 2 × 2
    □ ← 2 × 1 2 3
    □ ← 2 3 4 × 1 2 3
    □ ← 2 × 3 3⍴⍳9
    □ ← (3 3⍴⍳9) × 3 3⍴⍳9
    □ ← (1 2 3) (1 2 3) × (4 5 6) (5 6 7)
    □ ← 1 2 3 × double.PositiveInfinity
}
      fn()
4
2 4 6
2 6 12
  0  2  4
  6  8 10
 12 14 16
  0  1  4
  9 16 25
 36 49 64
 4 10 18  5 12 21
Infinity Infinity Infinity
```

The function can act as either a monadic or a dyadic primitive.

```
result ← expr1  expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dyadic functions are predefined for the integral types. For integral types and arrays of integrals, computes the logical NAND of its operands.

0 is always treated as false, all other values including 1 are treated as true.

Example

```
function fn() {
    □ ← 1 0 1 0 ⍲ 1 0 1 0
    □ ← 0 1 0 1 ⍲ 0 0 0 0
    □ ← 1 ⍲ 1
    □ ← 1 ⍲ 0
    □ ← 0 ⍲ 0
    □ ← 1 2 3 4 ⍲ 4 3 2 1
    □ ← 1 2 3 4 ⍲ 0 0 0 0
}

      fn()
0 1 0 1
1 1 1 1
0
1
1
0 0 0 0
1 1 1 1
```

The Natural Logarithm function can act as either a monadic or a dyadic primitive.

```
result ← ⍟ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Natural Logarithm function expands the Math.Log method to work with numeric arrays.

Math.Log Returns the natural (base **e**) logarithm of a specified number.

Example

```
function fn() {
    □ ← ⍟0
    □ ← ⍟1
    □ ← ⍟2.7182818284
    □ ← ⍟2.7182818284*2
}
¯Infinity
0
1
2
```

The Negate function can act as either a monadic or a dyadic primitive.

```
result ← – expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Negative function performs the negate operation on *expr1*. Negate (-) operators are predefined for all numeric types.

User-defined types can contain cross language overloads to the - operator.

Example

```
function fn() {
    □ ← –5
    □ ← –5 6 7
    □ ← –5 ¯6 ¯7
    □ ← –3 3⍴⍳9
}
      fn()
¯5
¯5 ¯6 ¯7
¯5 6 7
  0 ¯1 ¯2
 ¯3 ¯4 ¯5
 ¯6 ¯7 ¯8
```

The function can act as either a monadic or a dyadic primitive.

```
result ← expr1    expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dyadic functions are predefined for the integral types. For integral types and arrays of integrals, computes the logical NOR of its operands.

0 is always treated as false, all other values including 1 are treated as true.

Example

```
function fn() {
    □ ← 1 0 1 0 ⍱ 1 0 1 0
    □ ← 0 1 0 1 ⍱ 0 0 0 0
    □ ← 1 ⍱ 1
    □ ← 1 ⍱ 0
    □ ← 0 ⍱ 0
    □ ← 1 2 3 4 ⍱ 4 3 2 1
    □ ← 1 2 3 4 ⍱ 0 0 0 0
}

      fn()
0 1 0 1
1 0 1 0
0
0
1
0 0 0 0
0 0 0 0
```

The ~ function performs a logical NOT operation on its operand.

```
result ← ~ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

Monadic ~ functions are predefined for the number types. For number types and arrays of numbers, ~ computes the logical NOT of its operand.

0 is always treated as false, all other values including 1 are treated as true.

Example

```
function fn() {
    □ ← ~1 0 1 0
    □ ← ~0 0 0 0
    □ ← ~1
    □ ← ~0
    □ ← ~4 3 2 1
}

      fn()
0 1 0 1
1 1 1 1
0
1
0 0 0 0
```

The Not Approximately Equal function can act as either a monadic or dyadic primitive.

```
result ← expr1 ≠ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state: ⎕CT
```

The Not Approximately Equal function returns a 0 if *expr1* is equal to *expr2*, or if *expr2* is within ⎕CT of *expr1*. Otherwise, the return is 1.

Example

```
function fn() {
    ⎕ ← 10 ≠ 12
    ⎕ ← 10 ≠ 9 10 11
    ⎕ ← 10 ≠ 5+3 3⍴⍳9
    ⎕ ← 1 2 3 ≠ 1 2 3
    ⎕ ← 1 2 3 ≠ 1+1 2 3
    ⎕ ← 1 2 3 ≠ 1.1 2.1 2.1
    ⎕ ← (3 3⍴10.1) ≠ 3 3⍴10 11
}
      fn()
1
1 0 1
 1 1 1
 1 1 0
 1 1 1
0 0 0
1 1 1
1 1 1
 1 1 1
 1 1 1
 1 1 1
```

The ∨ function can act as either a monadic or a dyadic primitive.

```
result ← expr1 ∨ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dyadic ∨ functions are predefined for the integral types. For integral types and arrays of integrals, ∨ computes the logical OR of its operands.

0 is always treated as false, all other values including 1 are treated as true.

Example

```
function fn() {
    □ ← 1 0 1 0 ∨ 1 0 1 0
    □ ← 0 1 0 1 ∨ 0 0 0 0
    □ ← 1 ∨ 1
    □ ← 1 ∨ 0
    □ ← 0 ∨ 0
    □ ← 1 2 3 4 ∨ 4 3 2 1
    □ ← 1 2 3 4 ∨ 0 0 0 0
}

      fn()
1 0 1 0
0 1 0 1
1
1
0
1 1 1 1
1 1 1 1
```

The Outer Product function can act as either a monadic or dyadic primitive.

```
result ← expr1 ∘. operatorexpr1 expr2
```

Where:

result
An expression.

expr1
An expression.

operatorexpr1
An operator expression.

expr2
An expression.

Remarks

The Outer Product function is a specialized short hand construct simulating two nested for loops.

The Outer Product function creates its result by taking one element at a time from *expr1*, and calling the dyadic function specified by *operatorexpr1* with each element of *expr2*. Once the first element from *expr1* has been combined with every element from *expr2*, the next element from *expr1*, is taken, and the process is repeated, until each element of *expr1* has been combined with every element of *expr2*, through the dyadic operation specified in *operatorexpr1*.

Example

```
function fn() {
    ⎕ ← 'sample 1'              "         "
    ⎕ ← 1 ∘.+100 100 100
    ⎕ ← 'sample 2'              "         "
    ⎕ ← 10 10 10 ∘.+100 100 100
    ⎕ ← 'sample 3'              "         "
    ⎕ ← 11 12 13 ∘.+100 100 100
    ⎕ ← 'sample 4'              "         "
    ⎕ ← 11 12 13 ∘.+3 3⍴100 100 100
    ⎕ ← 'sample 5'              "         "
    ⎕ ← 11 12 13 ∘.+3 3⍴⍳9
}
      fn()
sample 1
 101 101 101
sample 2
 110 110 110
 110 110 110
 110 110 110
sample 3
 111 111 111
 112 112 112
 113 113 113
sample 4
 111 111 111
 111 111 111
 111 111 111

 112 112 112
 112 112 112
 112 112 112

 113 113 113
 113 113 113
 113 113 113
sample 5
 11 12 13
 14 15 16
 17 18 19

 12 13 14
 15 16 17
 18 19 20

 13 14 15
 16 17 18
 19 20 21
```

The Partition function can act as either a monadic or dyadic primitive.

```
result ←expr1 ⊂expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Partition function splits *expr2* into a nested vector, according to the enclosure pattern specified by *expr1*.

The rules for structuring an enclosure pattern are as follows:

- If an element of *expr1* is greater than (>) the previous element of *expr1*, than a new nesting group is begun, and the previous group is closed.
- If an element of *expr1* is less than or equal (<=) the previous element of *expr1*, then the corresponding element of *expr2* is included in the current nesting group.
- If an element of *expr1* is equal to 0, than the corresponding element of *expr2* is not included in the result.

Example

```
function fn() {
    a = 1 0 1 ⊂10 20 30
    ⎕←a
    ⎕←⍴a
    a = 1 0 1 ⊂3 3⍴⍳
    ⎕←a
    ⎕←⍴a
    a = 1 1 1 2 1 1 2 1 1  ⊂⍳
    ⎕←a
    ⎕←⍴a
    a = 1 1 1 2 1 1 ⊂2 6 ⍴⍳2
    ⎕←a
    ⎕←⍴a
}

      fn()
 10  30
2
 0 2
 3 5
 6 8
3 2
 0 1 2  3 4 5  6 7 8
3
 0 1 2   3 4 5
 6 7 8 9 10 11
2 2
```

The PiTimes function can act as either a monadic or dyadic primitive.

```
result ← ○ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The PiTimes function multiplies *expr1* by the system constant Math.PI.

At the time of this writing, the Math.PI system constant was held at: 3.14159265358979323846

Example

```
function fn() {
    □ ← ○1
    □ ← ○2
    □ ← ○1 2 ¯3
}
      fn()
3.141592654
6.283185307
3.141592654 6.283185307 ¯9.424777961
```

The Pick function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⊃ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state:   I O
```

The Pick function indexes into *expr2* at the index *expr1*, and discloses the result.

If the length of *expr1* is 1, *expr1* is used as an index into *expr2*, and the element produced from that index operation is then disclosed once, so that one level of nesting is removed from the element data.

If *expr2* has rank greater than 1, then *expr1* should contain an enclosed vector of indices, where the length of the vector is the same as the rank of *expr2*. Because Pick performs an index into *expr2* using the element from *expr1*, the enclosed vector can be any value that is valid for indexing into *expr2* using bracket indexing.

If the length of *expr1* is more than 1, a progressive Pick operation is performed. First, the last element of *expr1* is used to Pick data from *expr2*. Then, the next element of *expr1* is used to Pick data from the result returned by the first Pick. This continues until all elements of *expr1* have been processed. This functionality allows the short hand of only having to make a single call to the Pick function to perform a progressive Pick operation.

Example

```
function fn() {
    □ ← 1⊃1 2 3
    □ ← 2⊃(1 2 3) (4 5 6) (7 8 9)
    □ ← 1⊃2⊃(1 2 3) (4 5 6) (7 8 9)
    □ ← 1 2⊃(1 2 3) (4 5 6) (7 8 9)
    □ ← 1 2⊃ hello   world   more"     " "     " "    "
    □ ← (⊂(1 2) 2)⊃3 3⍴⍳9
}

      fn()
2
7 8 9
8
6
r
5 8
```

The Power function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⋆ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Power function returns *expr1* raised to the *expr2* power.

The Power function expands the Math.Pow method to work with numeric arrays.

Math.Pow returns a specified number raised to a specified power.

Note: For a complete and extensive list of how Math.Pow performs with special Double and Float values, such as Double.NaN and Double.PositiveInfinity, see the Math.Pow documentation available on Microsoft.com

Example

```
function fn() {
    □ ← 10 ⋆ 0
    □ ← 10 ⋆ 2
    □ ← 2.2 ⋆ 2
    □ ← 1 2 3 ⋆ 2
    □ ← 1 2 3 ⋆ 2 3 4
    □ ← (3 3⍴⍳9) ⋆ 2
    □ ← (3 3⍴⍳9) ⋆ 3 3⍴⍳9
}
      fn()
1
100
4.84
1 4 9
1 8 81
  0  1  4
  9 16 25
 36 49 64
     1       1         4
    27     256      3125
 46656  823543  16777216
```

The Ravel function can act as either a monadic or dyadic primitive.

```
result ← , expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Ravel function returns a vector which contains all elements of *expr1*, regardless of the shape of *expr1*.

If *expr1* is a scalar, the *result* vector has a length of 1 and contains 1 element.

If *expr1* is an array of rank 2, with 2 rows and 2 columns, the *result* has a length of 4 and contains 4 elements.

The Ravel function never changes the nesting level of *expr1*, as opposed to the Enlist function, which completely flattens an array, which includes removing all levels of nesting present in the data.

Example

```
function fn() {
    a = ,1
    □ ← a
    □ ← ⍴a
    a = ,1 2 3
    □ ← a
    □ ← ⍴a
    a = ,3 3⍴⍳9
    □ ← a
    □ ← ⍴a
    a = ,(1 2 3) (4 5 6)
    □ ← a
    □ ← ⍴a
}
      fn()
1
1
1 2 3
3
0 1 2 3 4 5 6 7 8
9
 1 2 3  4 5 6
2
```

The Reciprocal function can act as either a monadic or dyadic primitive.

```
result ←   expr1                                ÷
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Reciprocal function applies the mathematical reciprocal operation to its operand *expr1*, or 1 divided by *expr1*.

Example

```
function fn() {
    □ ←  1                                      ÷
    □ ←  1 2 3                                  ÷
    □ ←  3 3⍴1 2 3 4 5 6 7 8 9                  ÷
    □ ←   1 ¯2 ¯3                               ÷
}
      fn()
1
1 0.5 0.3333333333
            1   0.5 0.3333333333
         0.25   0.2 0.1666666667
 0.1428571429 0.125 0.1111111111
1 ¯0.5 ¯0.3333333333
```

Progressively performs the specified function between each element of *expr1*

```
return ← operatorexpr1 / expr1
return ← operatorexpr1   expr1
return ← expr2 operatorexpr1 / expr1
return ← expr2 operatorexpr1   expr1
```

Where:

result
An expression.

operatorexpr1
An operator expression.

expr1
An expression.

expr2
An expression.

Remarks

The Reduce function requires that *operatorexpr1* evaluate to a dyadic function to be a valid argument expression.

To see the effect of passing both *expr1* and *expr2* to the Reduce operator, please read below under: **Calling the Reduce operator dyadically**

Processing Order:

The Reduce operator is a specialized short hand construct simulating a single *for* loop, which progressively calls the dyadic *operatorexpr1* with the result of the last call to *operatorexpr1* as its right operand, and an element taken in receding order from the end of *expr1* as its left operand.

The Reduce function works exactly as a reverse *for* loop, where it iteratively calls a function with the result of the last iteration of the for loop as the right argument to the function, and the left argument is the next element in line from *expr1*. Note that the *for* loop is a reverse *for* loop in that it does not take elements from *expr1* starting at the first and proceeding to the last, but rather begins taking elements from end of *expr1*, until it reaches the first element.

Forms of Reduce:

There are two forms of the Reduce function:

/ (Reduce Last Dimension) and (Reduce First Dimension)

Both forms of Reduce perform exactly the same operation, except that they have a different default axis over which they apply the action on the data from *expr1*. These two forms of Reduce are provided as a short hand when processing data, since most data processing occurs on either the first of the last dimension of data. If an axis is explicitly specified, / (Reduce Last Dimension) and (Reduce First Dimension) perform exactly the same operations.

Calling the Reduce operator dyadically:

Because of the nature of the Reduce operator, only data from *expr1* is ever passed to the dyadic operator specified in *operatorexpr1.* With this being the case, data passed to Reduce through *expr2* is not used as the left argument in the call to *operatorexpr1*, but is rather an argument to the Reduce operator which denotes a special mode of processing the data in *expr1*. For more information on this mode of Reduce processing, please see: **Special Reduce Processing**.

Example

```
function fn() {
    □ ← +/1 2 3
    □ ← +/3 3⍴⍳9
    □ ← ×/1 2 3
    □ ← ×/3 3⍴⍳9
    □ ← 3+/1 2 3
    □ ← 3 3+/1 2 3 4 5 6 7 8 9 10 11 12
    □ ← 3 3+/2 12⍴1 2 3 4 5 6 7 8 9 10 11 12
}

      fn()
```

```
6
3 12 21
6
0 60 336
6
6 15 24 33
 6 15 24 33
 6 15 24 33
```

The Reshape function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍴ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Reshape function changes the shape of *expr2* to the shape specified in *expr1*, repeating or removing data as necessary.

expr1 should be an integral vector.

expr2 can be an array of any kind and shape.

If the number of elements required to fill an array of shape *expr1* exceeds the number of elements available in *expr2*, the elements of *expr2* are repeated as necessary, until all elements of the return array are filled.

If the number of elements required to fill an array of shape *expr1* is less than the number of elements present in *expr2*, than only as many elements as are needed to fill the result array are taken from *expr2*.

Following these definitions, if the number of elements required to fill an array of shape *expr1* matches the number of elements present in *expr2*, than no repeating or eliding of elements is performed.

Example

```
function fn() {
    □ ←  using shape to create a vector                    "
    a = 3⍴0
    □ ← a
    □ ← □dr a
    □ ←  using typing to create a vector                    "
    □ ←  creates vector with default value                    "
    □ ←  much quicker than shape                          "
    a = new int[3]
    □ ← a
    □ ← □dr a
    a = new double[3]
    □ ← a
    □ ← □dr a
    □ ←  create vectors with given values                    "
    □ ← 3 ⍴ 1
    □ ←  create 2 dimensional arrays                       "
    □ ← 3 3⍴⍳9
    □ ←  create 3 dimentional and n dimensional arrays                    "
    □ ← 3 3 3⍴⍳27
    □ ←  use nested arrays    "                "
    □ ← 3 ⍴ ⊂ test               "    "
    □ ← 3 ⍴  test  (1 2 3)       "    "
    □ ← 3 3 ⍴  test  (1 2 3)       "    "

}
      fn()
using shape to create a vector
0 0 0
323
using typing to create a vector
creates vector with default value
much quicker than shape
0 0 0
323
0 0 0
645
create vectors with given values
1 1 1
create 2 dimensional arrays
 0 1 2
```

```
 3  4  5
 6  7  8
create 3 dimentional and n dimensional arrays
  0   1   2
  3   4   5
  6   7   8

  9  10  11
 12  13  14
 15  16  17

 18  19  20
 21  22  23
 24  25  26
use nested arrays
 test  test  test
 test  1 2 3  test
test   1 2 3test
 1 2 3test   1 2 3
test   1 2 3test
```

The Residue function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍞ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Implicit Argument: ⎕CT
```

The residue operator (|) computes the remainder after dividing *expr2* by *expr1*.

Example

```
function fn() {
    ⍞ ← 10 | 10 11 12 20 21 22
    ⍞ ← 10 | 1 2 3
    ⍞ ← 10 11 12 | 10 11 12
    ⍞ ← 10 | 3 3⍴⍳9
    ⍞ ← (3 3⍴⍳9) | 3 3⍴⍳9
    ⍞ ← 10 | 10.1 10.2
}
      fn()
0 1 2 0 1 2
1 2 3
0 0 0
 0 1 2
 3 4 5
 6 7 8
 0 0 0
 0 0 0
 0 0 0
0.1 0.2
```

The Deal function can act as either a monadic or dyadic primitive.

```
result ← expr1 ? expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dependent state: ⎕IO, ⎕RL

Roll, monadic ?:

The Roll function selects a random integer between ⎕IO and (*expr2* - ⎕IO), for each element in *expr2*. *expr2* should evaluate to a single integer or an integer vector.

Deal, dyadic ?:

The Deal function creates a vector(s) of unique random integers, each equal in length to the each integer specified in *expr1*. For each element of *expr1*, the corresponding integer in *expr2* must be of a greater than or equal value.

Example

```
function fn() {
    ⍝ ← ?6
    ⍝ ← 6 ? 6
    ⍝ ← 6 6 ? 6
    ⍝ ← 6 6 ? 6 6
    ⍝ ← ? 3 3⍴6
    ⍝ ← 6 ? 2 2⍴6
    ⍝ ← 6 10 ? 6 10
    ⍝ ← ? 10 5 20 8
}
      fn()
5
5 3 4 0 2 1
 3 5 2 0 4 1  1 4 0 5 3 2
 5 0 4 1 3 2  0 3 1 4 2 5
 5 2 5
 0 4 0
 3 0 1
 2 5 1 4 0 3 4 0 5 2 3 1
 2 0 1 3 4 5 3 1 2 5 4 0
 1 0 3 2 4 5  0 7 1 6 4 5 2 8 3 9
1 0 11 4
```

The Rotate function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⌽ expr2
result ← expr1 ⊖ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Rotate function rotates the data supplied by *expr2* by the number if iterations specified by *expr1*.

The Reverse function, or the monadic form of Rotate, completely reverses the contents of *expr2*.

Dyadic Forms of Rotate:

The Rotate function has two dyadic forms:
-
⌽ (Rotate Last Dimension) and ⊖ (Rotate First Dimension)

The only difference between the two dyadic forms of Rotate is the default axis on which they rotate data in *expr2*. If the axis is explicitly specified, both forms produce the same result.

Monadic Forms of Reverse:

The Reverse function has two monadic forms:

⌽ (Reverse Last Dimension) and ⊖ (Reverse First Dimension)

The only difference between the two monadic forms of Reverse is the default axis on which they reverse data in *expr2*. If the axis is explicitly specified, both forms produce the same result.

Example

```
function fn1() {
    □ ← ⌽ hello world          "           "
    □ ← ⌽1 2 3 4.5 4.6 4.7
    □ ← ⌽3 3⍴⍳9
    □ ← 5 ⌽  hello world         "           "
    □ ← 1 ⌽ 3 3⍴⍳9
}
      fn1()
dlrow olleh
4.7 4.6 4.5 3 2 1
 2 1 0
 5 4 3
 8 7 6
 worldhello
 1 2 0
 4 5 3
 7 8 6


function fn2() {
    □ ←  rotate scalar        "             "
    □ ← ⊖1
    □ ←  rotate vector        "             "
    □ ← ⊖1 2 3
    □ ←  rotate matrix        "             "
    □ ← ⊖3 3 ⍴⍳9
    □ ←  specify amount to rotate axis                    "
    □ ← 1 2 ¯1 ⊖ 3 3⍴⍳9
    □ ← ⊖2 5⍴ helloworld          "           "
}
```

```
      fn2()
rotate scalar
1
rotate vector
3 2 1
rotate matrix
 6 7 8
 3 4 5
 0 1 2
specify amount to rotate axis
 3 7 8
 6 1 2
 0 4 5
world
hello
```

The Scan operator can act as either a monadic or dyadic primitive.

```
result ← operatorexpr1 \ expr1
```

Where:

result
An expression.

operatorexpr1
An operator expression.

expr1
An expression.

expr2
An expression.

Remarks

The Scan operator is a specialized short hand construct simulating a repeated call to the Reduce operator.

The Scan operator runs the Reduce operation on all element of *expr2*, then on (*expr2*.Length - 1) elements of *expr2*, then on (*expr2*.Length - 2) elements of *expr2.* Scan continues to decrement the number of elements on which it performs the Reduce operation, until there are no elements left across which to Reduce. The result of the Scan operation is the concatenated result of each call that was made to the Reduce operator during the Scan.

The result of each Scan operation is inserted into the result vector beginning at the last position and ending at the first, so that the result of the first Reduce operation is assigned into the last element of the return vector, and the last Reduce operation performed by the Scan is assigned to the first element of the result vector.

Example

```
function fn() {
    □ ← +\1
    □ ← +\⍳9
    □ ← +\3 3⍴⍳9
    □ ← 3+\1 2 3 4 5 6 7 8 9 10 11 12
    □ ← 3 3+\1 2 3 4 5 6 7 8 9 10 11 12
    a ← ,\ ab   cd   ed        "  " "  " "  "
    □ ← a
    a ← ,\2 6⍴10+⍳12
    □ ← a
}
      fn()
1
0 1 3 6 10 15 21 28 36
 0  1  3
 3  7 12
 6 13 21
6 9 12 15 18 21 24 27 30 33
6 15 24 33
 ab  cdab  edcdab
 10 10 11 10 11 12 10 11 12 13 10 11 12 13 14 10 11 12 13 14 15
 16 16 17 16 17 18 16 17 18 19 16 17 18 19 20 16 17 18 19 20 21
```

The Shape function can act as either a monadic or dyadic primitive.

```
result ← ⍴ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Shape function returns a vector of integers which are the current lengths of the dimensions of *expr2*.

Example

```
function fn() {
    □ ←  shape of scalar     "               "
    □ ← ⍴1
    □ ←  shape of vector     "               "
    □ ← ⍴,1
    □ ← ⍴1 2 3
    □ ← ⍴3 3⍴⍳9
    □ ← ⍴1  abc  (2 3 4)  more  "   "         "    "
}
      fn()
shape of scalar

shape of vector
1
3
3 3
4
```

The Sign function can act as either a monadic or dyadic primitive.

```
result ← × expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

Returns a value indicating the sign of a number, where a negative number has a sign of -1, a positive number has a sign of 1, and a 0 has a sign of 0.

Example

```
function fn() {
    □ ← × 10
    □ ← × 0
    □ ← × ¯10
    □ ← × 10 0 ¯10
    □ ← × 3 3⍴10 0 ¯10
}


      fn()
1
0
¯1
1 0 ¯1
 1 0 ¯1
 1 0 ¯1
 1 0 ¯1
```

The Squad Index function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⌷ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Provides a primitive for indexing.

expr1 is any array which is valid for bracket indexing.

Example

```
function fn() {
    a = 1 2 3 4
    ⎕ ← 'index a vector with a scalar'
    ⎕ ← 1 ⌷ a
    ⎕ ← 'index a vector with a vector'
    ⎕ ← (1 2) ⌷ a
    a = 3 3⍴⍳9
    ⎕ ← 'index a matrix with a vector'
    ⎕ ← 1 1 ⌷ a
    ⎕ ← 'index a matrix specifying axis'
    ⎕ ← 1 ⌷ [1] a
    ⎕ ← 'index a matrix with a vector'
    ⎕ ← (1 2) ⌷ a
    ⎕ ← 'index a matrix with a vector and scalar'
    ⎕ ← (1 2) 1 ⌷ a
    ⎕ ← 'index a matrix with two vectors'
    ⎕ ← (1 2) (,1) ⌷ a
}
      fn()
index a vector with a scalar
2
index a vector with a vector
2 3
index a matrix with a vector
4
index a matrix specifying axis
1 4 7
index a matrix with a vector
5
index a matrix with a vector and scalar
4 7
index a matrix with two vectors
 4
 7
```

The Subtract function can act as either a monadic or dyadic primitive.

```
result ← expr1 – expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Subtract functions subtract the second operand from the first. Subtract functions are predefined for all numeric and enumeration types

User-defined types can contain cross language overloads to the - operator.

Example

```
function fn() {
    □ ← 2 – 1
    □ ← 2 – 1 2 3
    □ ← 1 2 3 – 1 2 3
    □ ← 1.1 1.2 1.3 – 1
    □ ← 1.1 1.2 1.3 – 1.1 1.2 1.3
    □ ← 1 – 3 3⍴⍳9
    □ ← (3 3⍴⍳9) – 3 3⍴⍳9
}
      fn()
1
1 0 ¯1
0 0 0
0.1 0.2 0.3
0 0 0
  1  0 ¯1
 ¯2 ¯3 ¯4
 ¯5 ¯6 ¯7
 0 0 0
 0 0 0
 0 0 0
```

The Take function can act as either a monadic or dyadic primitive.

```
result ← expr1 ↑ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Take function returns data from dimensions of *expr2*, according to the amounts specified in *expr1*.

The length of *expr1* should match the rank of *expr2*, and each element of *expr1* specifies the amount of data to Take from the respective dimension of *expr2*.

The elements of *expr1* can be either negative, positive, or 0. If an element of *expr1* is positive, that length is taken from the related dimension of *expr2*. If an element of *expr1* is negative, that length is taken from opposite end of the related dimension of *expr2*. If an element of *expr1* is 0, the data is elided from the resultant dimension of the result.

Example

```
function fn() {
    □ ← 1 ↑ 10
    □ ← 2 ↑ 10
    □ ← 2 ↑  a                    " "
    □ ← 10 ↑ 10
    □ ← 2 2 ↑ 3 3⍴⍳9
    □ ← ¯2 ¯2 ↑ 3 3⍴⍳9
    □ ← 4 ↑ (1 2) (3 4)
}
      fn()
10
10 0
a
10 0 0 0 0 0 0 0 0 0
 0 1
 3 4
 4 5
 7 8
 1 2  3 4  0 0  0 0
```

Produces a single number of radix base 10 from expr2, where expr2 is a vector of numbers, and expr1 is a vector of numbers specifying the radix of each element of expr2.

```
result ← expr1 ⊥ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

If *expr1* is a scalar, *expr1* is considered to be the same length as *expr2* (scalar expansion).

Example

```
function fn() {
    □ ← 10 10 10 10 ⊥ 1 7 7 6
    □ ←  Convert 2 days, 12 hours, 22 minutes to total minutes                    "
    □ ← 1 24 60 ⊥ 2 12 22
    □ ←  Convert 8 bits to base 10 number                    "
    □ ← 2 2 2 2 2 2 2 2 ⊥ 0 0 0 0 1 0 1 0
}
      fn()
1776
Convert 2 days, 12 hours, 22 minutes to total minutes
3622
Convert 8 bits to base 10 number
10
```

The Transpose function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍉ expr2
result ← ⍉ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dyadic Transpose:

The Transpose function creates a result array that contains all elements of *expr2*, except that the dimensions of the data, and consequently the positions of the data in the *result* array, are remapped according to the *remap* sequence specified by *expr1*.

The length of *expr1* must be equal to the rank of *expr2*.

expr1 must be a vector of indices, where no index is greater than the rank of *expr2.*

If all elements of *expr1* are unique, then following definition of Transpose applies:

The *result* of Transpose is obtained by iterating sequentially through each element of *expr2*, determining the array index if that element, remapping that array index according to *expr1*, and then assigning the indexed element into the result array at the remapped index.

If elements of *expr1* are repeated, then the following definition applies:

The elements of the *result* of Transpose are the elements in *expr2* where the following definition holds true:

```
An element is selected from expr2, where the array index of that element has repeated
indices at the same locations as the repeated indices in expr1.
```

Monadic Transpose:

If the left argument to the Transpose function is omitted, the dimensions of *expr2* are reversed. The result of Monadic Transpose can be replicated with dyadic Transpose, if the supplied *expr1* is a reversed vector of indices from 1 to the rank of *expr2*.

Example

```
function fn() {
    □ ← ⍉1
    □ ← ⍉1 2 3
    □ ← ⍉2 4⍴⍳8
    □ ←  specify axis        "            "
    □ ← 0 1⍉2 4⍴⍳8
    □ ←  reorder axis        "            "
    □ ← 1 0⍉2 4⍴⍳8
    □ ←  reorder axis        "            "
    □ ← 2 0 1⍉2 4 2⍴⍳16
}
      fn()
1
1 2 3
 0 4
 1 5
 2 6
 3 7
specify axis
 0 1 2 3
 4 5 6 7
reorder axis
 0 4
```

```
 1 5
 2 6
 3 7
reorder axis
 0  8
 1  9

 2 10
 3 11

 4 12
 5 13

 6 14
 7 15
```

The Trigonometric function can act as either a monadic or dyadic primitive.

```
result ← expr1 ○ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

This primitive provides array extensions to all of the System.Math libraries, and also provides additional functionallity not found on System.Math.

Valid expr1 elements and their meaning are:

```
¯7 - Hyperbolic Arc Tan
¯6 - Hyperbolic Arc Cos
¯5 - Hyperbolic Arc Sin
¯4 - (¯1+expr2*2)*0.5
¯3 - Arc Tan
¯2 - Arc Cos
¯1 - Arc Sin
 0 - (1-expr2*2)*0.5
 1 - Sin
 2 - Cos
 3 - Tan
 4 - (1+expr2*2)*0.5
 5 - Hyperbolic Sin
 6 - Hyperbolic Cos
 7 - Hyperbolic Tan
```

expr1 can be either a scalar or array, and is applied to *expr2*.

Example

```
function fn() {
    □ ← 0 2 ○ .5 .5
    □ ← 1 ○ .5
}
      fn()
0.8660254038 0.8775825619
0.4794255386
```

Dyadic function ~ evaluates whether the elements in expr1 exist in expr2, and returns those elements of expr1 which do not exit in expr2.

```
result ← expr1 ~ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

`Dependent state: ⎕CT`

Dyadic function ~ evaluates whether the elements in *expr1* exist in *expr2*, and returns those elements of *expr1* which do not exit in *expr2*.

Example

```
function fn() {
    ⍝ ← 1 2 3 ~ 1 2 3 4 5 6
    ⍝ ← 1 2 3 4 5 6 ~ 1 2 3
    ⍝ ← 1 ~ 2
    ⍝ ← 1 ~ 1
    ⍝ ←  test   two  ~  test  " three "   "   "    " "     "
}

      fn()

4 5 6
1

 two
```

Visual Studio .NET Tips and Tricks

Abstract

This book provides you with the information you need to effectively use Visual APL in the Visual Studio development environment.

Navigation

To navigate this book, you may either use the tree view on the left to navigate chapter by chapter, section by section, or view the entire book on one page in "Visual Studio .NET Tips and Tricks" in the left hand tree view.

Introduction

Visual Studio .NET comes complete with many features and functions that dramatically increase our efficiency as developers. As a powerful code editor, compiler, and debugger, it contains features to stress-test, analyze, and optimize your code, and allows easy integration with code documentation, reporting, or smart-device programming, such as the Pocket PC.

Because of the sheer number of features that Visual Studio .NET contains, it is a challenge for .NET developers to become familiar with all of its features, shortcuts, and functionalities. The beginner developer will find a virtual treasure trove of features with which to start, while advanced Visual Studio .NET users will appreciate the many new features and improvements the new Visual Studio .NET 2005 brings.

Most of these features and functionalities are documented, and are accessible through the VS.NET main menu or context menus. But, because of the vast number of features with which VS.NET is equipped, developers don't always know them or use them. This guide should help familiarize developers with the tips and tricks that are at their disposal in this powerful tool and their specific application to the Visual APL language.

Chapter 1: Editing Code

While you create code, there are many techniques and shortcuts that allow you to write and navigate through your code quickly and easily. This chapter introduces many of the tips and tricks you will need for such tasks as code navigation, performing complex find-and-replace searches, and generating code.

1.0 - Inserting Comment Tokens (Ctrl-K,Ctrl-H)

This feature enables you to write a comment and find it again easily. To see a list of all reminders you placed in your code (see Figure 1):

> Without recompiling, select View > Show Tasks > All

To insert task shortcuts in your code:

> Press Ctrl-K, Ctrl-H.

This marks the current line with a shortcut icon and inserts a clickable shortcut icon in the Task List.

To remove the shortcut:

> Press Ctrl-K, Ctrl-H.

These shortcuts survive IDE restarts.

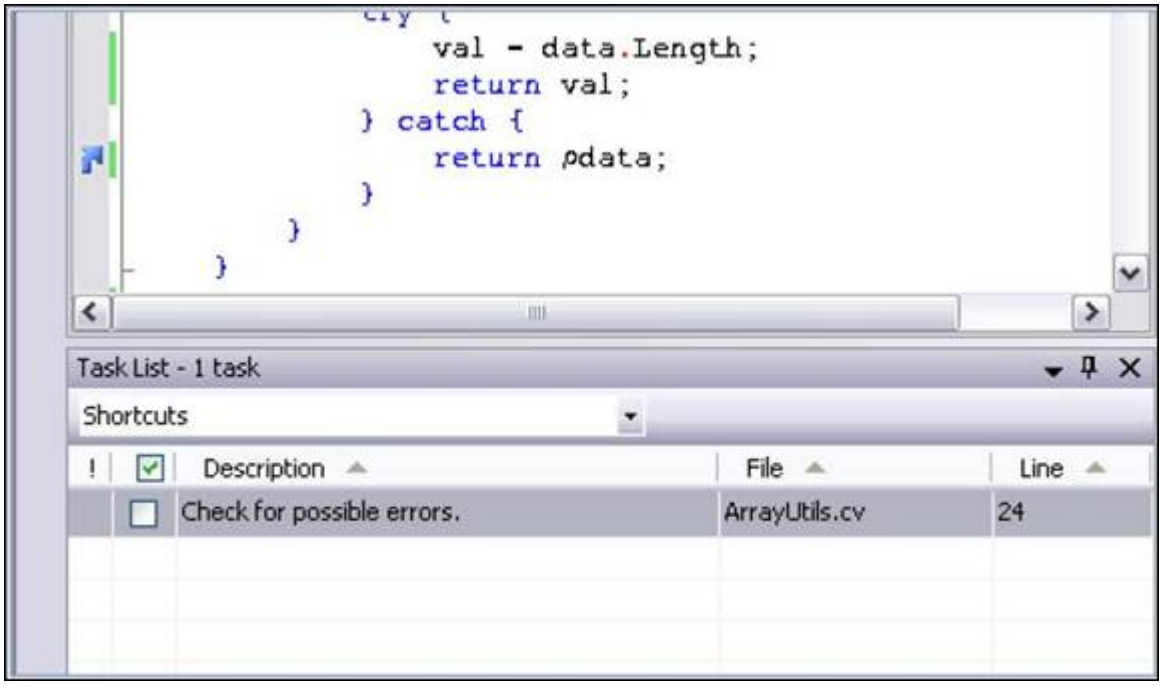


Figure 1. Comment Tokens.

1.1 - Commenting Code Blocks (Ctrl-K, Ctrl-C)

One-line comments are extremely useful in explaining seldom used code and assisting in navigation and definition of development projects. To inserted a comment for a code block or segment:

Press the "//" token for Visual APL.

Additionally, Visual APL allows you to comment entire paragraphs and segments. To place a comment in a paragraph or segment:

Select "/#" (and corresponding "#/") tag around the comment.

To quickly comment entire paragraphs:

Select the text.

Click the Comment button (see Figure 2) or

Press Ctrl-K, Ctrl-C.

This comments an entire selection.

To uncomment any selection:

Click the Uncomment button or

Press Ctrl-K, Ctrl-U.

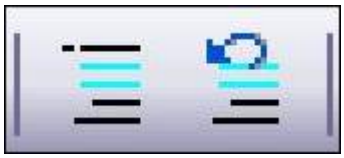


Figure 5 - Comment and Uncomment buttons

1.2 -Creating Regions

The more code you generate, the more difficult it can become to navigate. In addition to selecting classes and their methods from the drop-down lists above the main editor, you can also group your code into logical regions. Regions are extremely helpful for dividing code in logical ways and even commenting it. Regions allow you to collapse code to a single line defining the region and still easily see what is inside it once it is collapsed. They can even be nested. Automatically generated code in VS.NET usually uses this feature, so you may already be familiar with it.

To specify a region:

Insert a #region keyword and a description at the beginning of your segment and a corresponding #endregion keyword at the end of your segment (see Figure 3).

```
#region Using directives

using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

#endregion
```

Figure 3 - Creating regions

The Outlining menu displays various collapse and expand options. To expand and collapse the current region you are in:

Press Ctrl-M, Ctrl-M.

To expand or collapse all regions at once:

Right-click the gray bar to the left of the main editor window.

To collapse an individual region:

Click the plus sign next to the #region keyword.

This collapses the code into a single line that shows the region description.

To display the inside of a collapsed region:

Move the mouse over the gray description area (see Figure 4).

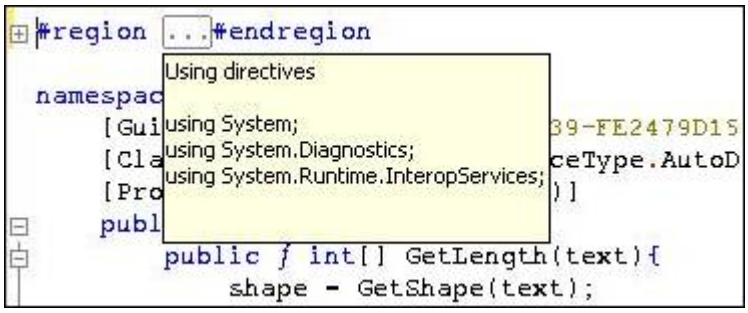


Figure 4 - Mouse over a region to see its content

You can even drag and drop collapsed regions inside your code. When you paste a collapsed region into a different location, the pasted text is automatically expanded.

1.3 -Hiding Selection by Using Temporary Regions (Ctrl-M,Ctrl-H)

Regions are created automatically for methods, comments, and sections encompassed by the #region compiler directive. In Visual APL and in regular text files, you have the option to create temporary regions around any section without the need for "#region". This is useful when you want to create a region that will not be preserved once your project is closed. In this case, you can temporarily define a region using the following method:

Highlight the section you want to hide and press Ctrl-M, Ctrl-H.

This hides the current selection in a temporary collapsed region (see Figure 5).

To expand a temporary collapsed region:

Press Ctrl-M, Ctrl-M.

Temporary regions are lost after you close a project.

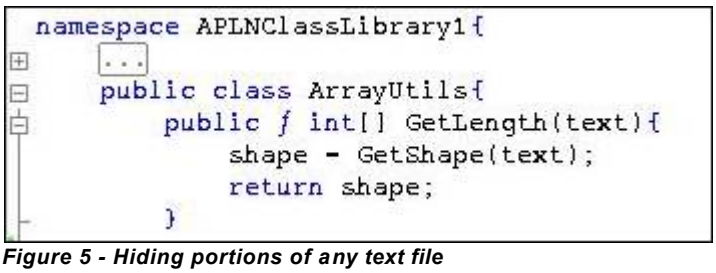


Figure 5 - Hiding portions of any text file

For creating regions which are preserved past the closing of your project, see "Creating Regions"

1.4 -Selecting a Single Word (Ctrl+W)

To select a single word when editing code:

Double-click anywhere in the word or just press Ctrl-W.

Double-clicking in a word is a common method used by many word processing and publishing programs to quickly select a word.

1.5 -Placing Code into the Toolbox (Ctrl-Alt-X)

When creating a project, you may want to use certain pieces of code or text again and again. You may have a standard copyright header that you place at the top of each file or a certain line of code to perform a common task. To simplify this repetitive task, you can place it into your Toolbox. The Toolbox is the window that lists all windows or web controls. To place your item into the Toolbox, use the following method:

1. Pull up the Toolbox:

   Press Ctrl-Alt-X.

2. Move the frequently used text or code into the Toolbox:

   Highlight your item and drag the selected text onto the General tab in your Toolbox (see Figure 10).

3. Rename the produced text item in your Toolbox

   Right-click it and choose Rename Item from the pop-up menu.

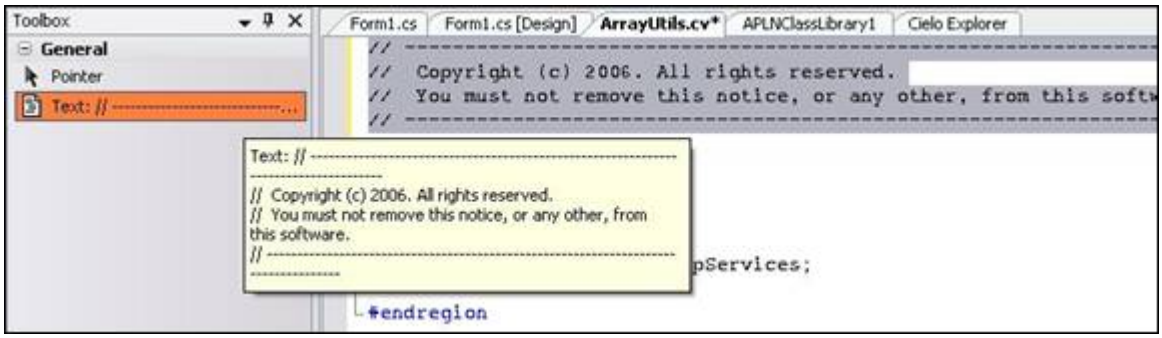


Figure 10. Adding text to the Toolbox

To insert an item into your text or code from the Toolbox:

Select the item in your Toolbox, then either:

Drag it into your code window or

Place your cursor in your text where you want to insert the item,
Double-click the entry

The General tab in the Toolbox is project and solution independent, and it retains its content even after you restart VS.NET.

1.6 - Using the Clipboard Ring (Ctrl-Shift-V)

The Clipboard Ring works like a historical file of the last used text selections that you placed on the Clipboard. Because it preserves many levels of selections, it is useful when you accidentally overwrite the current Clipboard content or when you find yourself needing several different items concurrently.

To use the Clipboard Ring you can either:

Double-click one of the remembered Clipboard items to paste it at the cursor's current location or

Drag it into the editor.

When the Clipboard Ring contains many Clipboard items, or when you cannot see the complete contents of each item in the ring, it's useful to cycle through the Clipboard Ring.

To progress one item at a time through the Clipboard Ring:

Select Edit > Cycle Clipboard Ring (Ctrl-Shift-V)

Doing this repeatedly makes VS.NET cycle through the Clipboard Ring's contents, displaying the stored Clipboard contents in the text editor at the cursor's current location. This method makes it easy to paste specific content in the code editor as it becomes visible during the cycling. Continue cycling through the Clipboard's contents until you find your desired item.

1.7 - Transposing a Single Character or Word (Ctrl-T or Ctrl-Shift-T)

To switch the position of the current characters or words on either side of the cursor you need to use Transpose. This procedure switches the characters or words, then moves the cursor to the right. Transpose is useful if you mistype a word or write a sentence or code segment with words in the incorrect order.

To transpose a single character:

Press Ctrl-T.

This swaps the two characters surrounding the cursor and moves the cursor to the right by one character. Pressing Ctrl-T repeatedly allows you to move a single character further to the right one character at a time.

To transpose a single word:

Press Ctrl-Shift-T.

Note: This does more than just swap two adjacent words. VS.NET knows to ignore "unimportant" single characters, such as equal signs, string quotes, white spaces, commas, etc.

Suppose you have a line of code that originally looks like this:

```
new SqlCommand( trans , stored_procedure, conn);
```

Pressing Ctrl-Shift-T repeatedly on the word "trans" would yield the following:

```
new SqlCommand( stored_procedure , trans, conn);        "
```

and finally this:

```
new SqlCommand( stored_procedure , conn, trans);        "
```

The quotation marks and commas retain their original positions throughout the process. When you reach the end of a line, pressing Ctrl-Shift-T transposes the word with the first word of the next line.

1.8 - Cutting, Copying, Deleting, and Transposing a Single Line

If you need to cut, copy, delete or transpose an entire line, you can do this easily with one keyboard sequence.

To **copy** the complete, current line to the Clipboard:

Press Ctrl-C (or click the Copy icon) without any text selected.

To **cut** an entire line:

Press Ctrl-X (or click the Cut icon) without any text selected.

This will cut the entire current line and place it in the Clipboard.

To **delete** a single line:

Press Ctrl-L without any text selected.

To **transpose**, or **swap** the current line with the one below it:

Press Alt-Shift-T without any text selected.

Doing this also moves the cursor down by one line. This allows you to press this keyboard shortcut repeatedly until you move your current line to the desired position.

1.9 - Formatting Entire Blocks (Ctrl-K, Ctrl-F or Ctrl-K,Ctrl-D)

To apply formatting to an entire selection, there are several useful functions you can use. Uppercasing, lowercasing, or deleting horizontal white spaces are just a few examples.

To access these features:

Select Edit > Advanced

One of the most useful features here is the Format Selection function.

To access the Format Selection Function:

Press Ctrl-K, Ctrl-F,

This feature formats an entire selection and inserts tabs where appropriate to modify the code with the correct code-specific block indentation. This is usually done automatically when someone enters code upon closing a block (such as by typing the "}" sign in Visual APL) but Format Selection forces this automatic format (see Figures 11 and 12).

You can also format the entire document. To do this:

Press Ctrl-K, Ctrl-D.

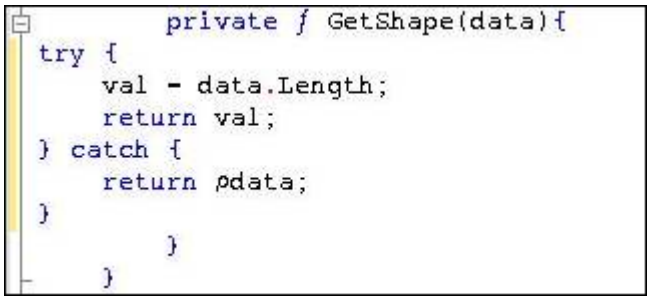


Figure 11. Before formatting block

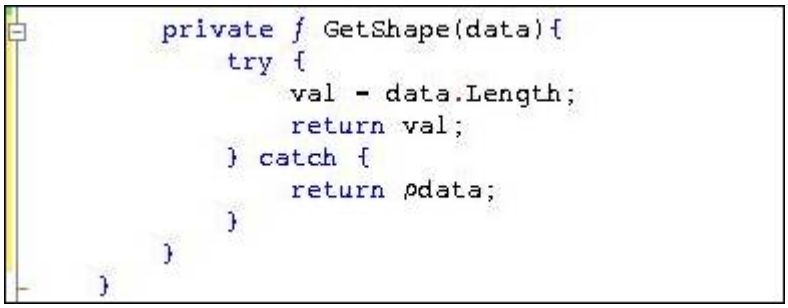


Figure 12. After formatting block

1.10 - Toggling Word-Wrapping (Ctrl-R,Ctrl-R)

To turn word wrapping on and off for the current view:

Select Edit > Advanced

Or use the keyboard shortcut:

(Ctrl-R, Ctrl-R)

1.11 - Creating GUIDs

As you develop new classes and components, you often need to create Global Unique Identifiers (GUIDs). These are 128-bit values often represented by 32 hexadecimals. In the past, component developers used GUIDs to assign their components with unique names to reduce the likelihood of two components sharing the same GUID. Developers now use GUIDs for anything that requires a unique identifier. GUIDs can be created manually by randomly selecting 32 hexadecimals, but this is somewhat tedious. VS.NET comes with a utility that creates GUIDs for you whenever you need one.

To create a GUID, open the Create GUID dialog box:

Select Tools > Create GUID (see Figure 13).

Here you can generate identifiers in various formats, including common code items often used in COM development.

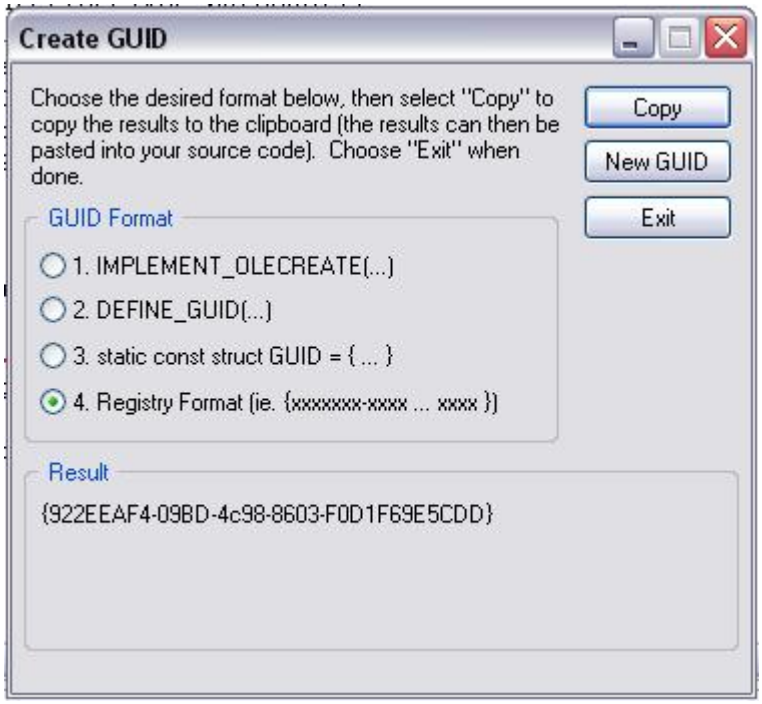


Figure 13. Create GUID tool

1.12 - Creating Rectangular Selections

To make a rectangular selection of text or code, there is a technique which allows you to do this without including the intervening lines (see Figure 14).

To select a rectangular area:

Press the Alt key while dragging the mouse to select the area.

Manipulating the selection by copying, cutting, or pasting rectangular blocks can be done very easily this way.

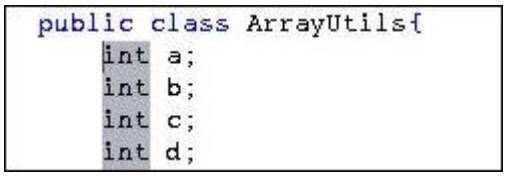


Figure 14. Rectangular selection

1.13 - Switching Between Views (F7)

For Windows forms, you can easily switch between both views. To toggle between designer and code views:

Press F7 (designer and code)

1.14 - Going to a Line Number (Ctrl-G)

For quick and easy navigation inside your code or text file, you can jump to a particular line.

To go to a specific line number you need to access the go to dialog box. To do this, either:

press Ctrl-G or

double-click the line number status bar at the bottom.

A small dialog box will appears. To jump to a line number:

enter a line number

If you type a line number that is out of the range of possible line numbers, the cursor jumps to the beginning or end of the file, respectively. A number which exceeds the number of lines in the file will place you at the end of the file. A number that is too low will jump you to the beginning of the file.

1.15 - Searching for a Word

There are several methods for searching for a word inside a file. It is helpful to know all the methods for ease in moving around your file. The following are common methods for finding a word.

1. Access the Find dialog box

   Select Edit > Find

   Enter a term in the Find dialog box.

2. Access the Combo box in the main toolbar next to the configuration drop-down list. To open the combo box and invoke the search function:

   Press Ctrl-D

   Enter or paste a word into this list and press Enter

   Repeat pressing Enter in that drop-down list to find the next match.

3. Select the entire word, or place the cursor somewhere inside the word:

   Press Ctrl-F3.

   This invokes the same search function I described just previously. Repeatedly pressing Ctrl-F3 iterates over all matches.

Using either the combo box or the Ctrl-F3 shortcut applies the same search options specified in the Find dialog box. Set the options you desire in the Find dialog box first to search correctly (for example, enabling Search Hidden Text to include all collapsed regions in the search area).

1.16 - Performing an Incremental Search (Ctrl-I)

An incremental search allows you to find occurrences of a search key as you type it one letter at a time. After each keystroke, VS.NET immediately highlights the next available occurrence that matches whatever you have typed so far. The more letters you type, the more likely is it that the found occurrence is indeed what you are seeking.

To initiate an incremental search:

Press Ctrl-I

You do not need to enter the entire word to find a specific occurrence; you only need to type the minimum number of characters that would uniquely identify the word for which you are searching.

To return to normal editing mode:

Press Escape

In the Cielo Explorer you have to single click with the mouse.

If you are unsatisfied, press Ctrl-I repeatedly to find the next occurrence that matches your partial search key, or press Ctrl-Shift-I to find the previous matches. You can, of course, simply enter more letters to narrow the search further.

1.17 - Searching or Replacing with Regular Expressions or Wildcards

Regular expressions can look extremely intimidating, but they are extremely powerful tools to find complicated search keys and patterns. Regular expressions is a built in feature that allows you to describe a

searchable pattern in terms of wildcards, characters, and groups.

This feature in VS.NET is often overlooked by many developers. To access this feature, bring up the Search or the Replace dialog box:

Press either Ctrl-F or Ctrl-H, respectively

Note: Besides the regular options to refine your search, the last check box allows you to define your search based on regular expressions or wildcards.

You can use either of two modes. To use Regular Expression mode, specify the expression using a similar notation you are accustomed to with the System.Text.RegularExpressions namespace.

To see a list of possible constructs that you can insert into your regular expression:

Click the arrow button next to the Find What field (see Figure 15).

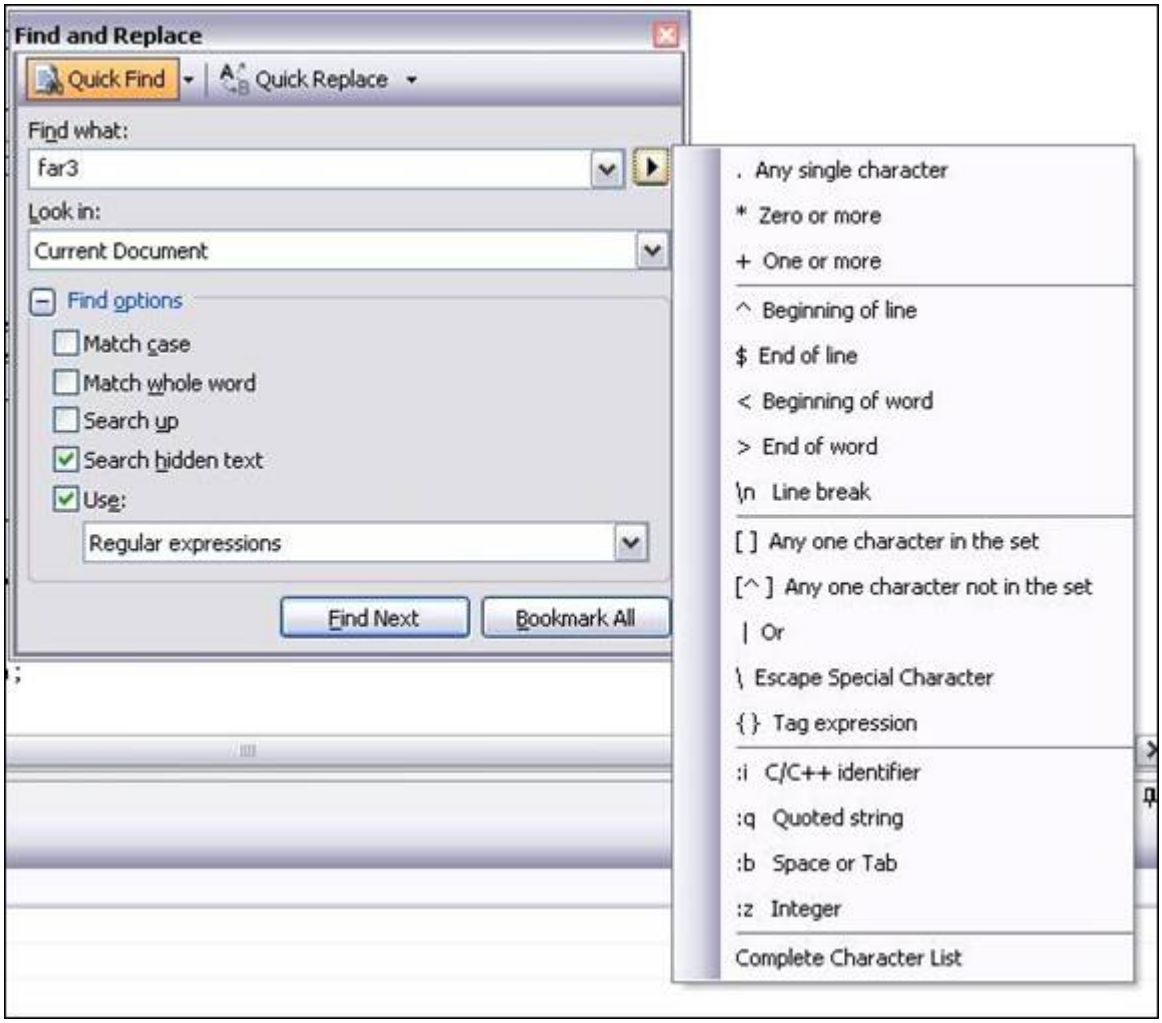


Figure 15. Use regular expressions.

To use Wildcard mode, construct your search pattern using the more commonly known wildcards from MS-DOS, such as "*" and "?".

If used correctly, these two modes can be very helpful in refining your search algorithm or when developing programs based on regular expressions.

1.18 - Global Search or Replace (Ctrl-Shift-F or Ctrl-Shift-H)

The global search and replace feature in VS.NET spans entire projects and solutions. This is similar to normal search and replace dialog boxes, except that you can specify the scope of the search or replace action over multiple files.

To bring up the global search or global replace dialog box:

Press Ctrl-Shift-F or Ctrl-Shift-H, respectively

This feature allows you to perform a global search and replace in just the current document, the current project, the entire solution, or any open documents (see Figure 16). You can also filter which files you want to search based on wildcards.

Figure 16. Global search and replace.

Once the search or replace action is started, VS.NET searches all specified documents and modifies them if required. The global replace, will also prompt you to leave modified documents open. This option allows you to undo the replace, because only open documents offer the undo feature. If you don't select that option, global replace will automatically save the modified files and make this a permanent action.

To immediately stop a global search or replace anytime:

> Press Ctrl-Break.

Once a search or replace action is completed, a list of occurrences that have been found will be displayed in the "Find Result" dialog box.

To iterate over the Find Result list:

> Press F8 or navigate to an occurrence by double-clicking it.

If that occurrence is currently located in a collapsed region, you can expand it. To automatically expand the region:

> Double-click the same find result in the list again.

On initiating a new find or replace action, VS.NET clears this window to fill the list with the new results. If you want to keep the results of the previous search and output the result in a second window:

> Check the Display in Find 2 option in the search dialog box

You can then tab between both result sets.

All find/replace functionalities are included in a single dialog box (see Figure 17), you can also access the global find/replace functionalities using the drop-down list at the top. All shortcuts remain the same.

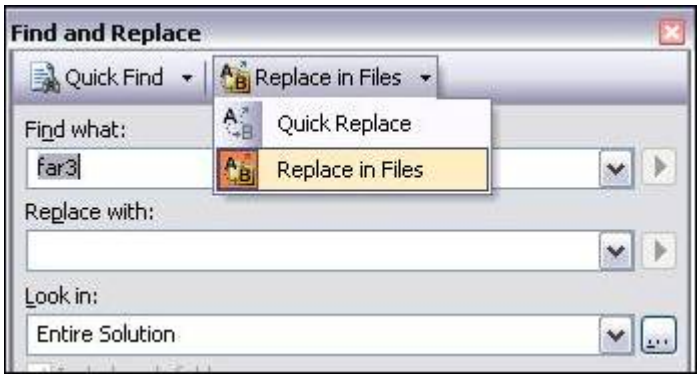


Figure 17. Global replace in files

1.19 - Using Bookmarks

Bookmarks enable you to return quickly to a given page or section of your code or file. When you determine critical sections of your programming that you want to return to frequently, instead of scrolling to these places, bookmark those lines.

To place a bookmark, first, make the bookmark toolbar visible:

Right-click any existing toolbar and select Text Editor from the pop-up menu.

Click on the blue flag icon in Text Editor toolbar.

Another method which can be used to place a bookmark:

Press Ctrl-K, Ctrl-K.

This second method not only makes a bookmark visible on the left side of the code, but you can now jump quickly among other bookmarks. To jump to the other bookmarks you can either:

Click the appropriate flag buttons on the toolbar (see Figure 18) or

Press Ctrl-K, Ctrl-P (for the previous bookmark) or Ctrl-K, Ctrl-N (for the next bookmark).

Figure 18. Bookmark toolbar

To clear all bookmarks:

Press the Clear Flag icon or

Press Ctrl-K, Ctrl-L.

The Find dialog box in VS.NET allows you to bookmark all matching occurrences as follows:

Click the Mark All button.

As part of VS.NET 2005 considerable support for bookmarks, you can also have the option of moving to the next or previous bookmark within the same file as follows:

Press the appropriate buttons on the bookmark toolbar (see Figure 28).

Figure 28 - Move to bookmarks in the same file in VS.NET 2005

You can also name your bookmarks by first opening a new Bookmarks window:

Press Ctrl-K, Ctrl-W or

Select View > Other Windows >Bookmark Window.

This displays all the bookmarks that you have created (see Figure 19).

To jump to a bookmark's location:

Double-click the bookmark.

To rename a bookmark:

Press F2 or

Right-click the bookmark and use the Rename context menu item.

You can categorize your bookmarks and organize them into folders. You can also, jump to the next or previous bookmark within the same folder. To perform any of these functions, simply select the appropriate icon on the toolbar.

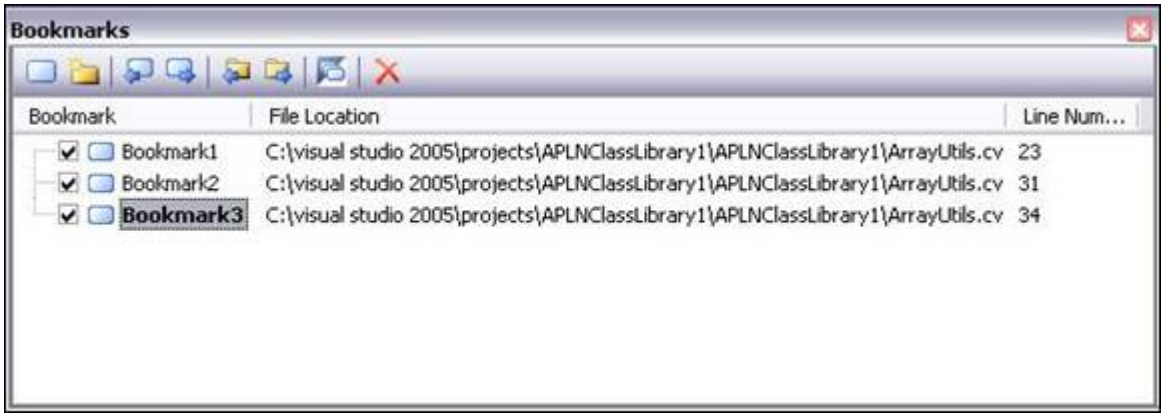


Figure 19. Manage your bookmarks in the Bookmarks Window.

The Bookmarks window shows check boxes next to each folder and bookmark. These allow you to disable a bookmark without deleting it. Disabled bookmarks are skipped when you use any of the buttons or shortcuts to navigate your bookmarks.

1.20 - Using Browser-Like Navigation (Ctrl -, Ctrl Shift -)

VS.NET is equipped with browser-like "back" and "forward" buttons in the IDE that allow you to review the most recent cursor locations. The Navigate-Backward and Navigate-Forward buttons are located to the right of the Undo and Redo buttons (see top left of Figure 20). You can also access them in the View menu.

Figure 20. Navigate buttons

Similar to a web browser, VS.NET keeps a history of your recently accessed locations. After using the Go To Definition feature or after switching arbitrarily to another file or even just jumping between different line numbers of the same file, you can easily return back to the last edit location as follows:

Click the Navigate-Backward button.

These Navigate buttons have pre-assigned shortcuts. To Navigate back:

Press Ctrl-Hyphen

To Navigate forward:

Press Ctrl-Shift-Hyphen.

1.21 - Inserting External Text File

A common method of inserting code fragments involves opening a file in Notepad and copying the code from there. To bypass this step of opening and closing Notepad you can:

Select Edit > Insert File as Text from within the code editor.

Chapter 2: Exploring the IDE

Visual Studio .NET is an easily customizable feature-rich Integrated Development Environment (IDE). It allows a developer quick access to commonly used commands and activities which enable you to control and modify your project and solutions. This chapter covers a range of topics such as: the Solution Explorer; window positioning; managing macros; modifying menu items and other tips and tricks useful for in navigating inside the IDE.

2.0 - Setting Project Dependencies

In a large solution with multiple projects and custom build events, it is often necessary to control the build order for your projects. VS.NET has the capability to figure out which project needs to be built first by analyzing the references of each one. The first project built is normally the one referenced first. This algorithm is based on your set project references for your projects.

VS.NET also allows you to compile a certain project before another one without having project references. This is accomplished from a pop-up menu that allows you to choose Project Dependencies. To designate the order in which projects will be built:

Right-click your project that needs to be built last and choose Project Dependencies from the pop-up menu.

Set manual dependencies on other projects by check-marking them.

This will ensure that the checked projects will be built before the current project (see Figure 21). A drop-down menu for the current project allows you to switch to another project's dependencies.

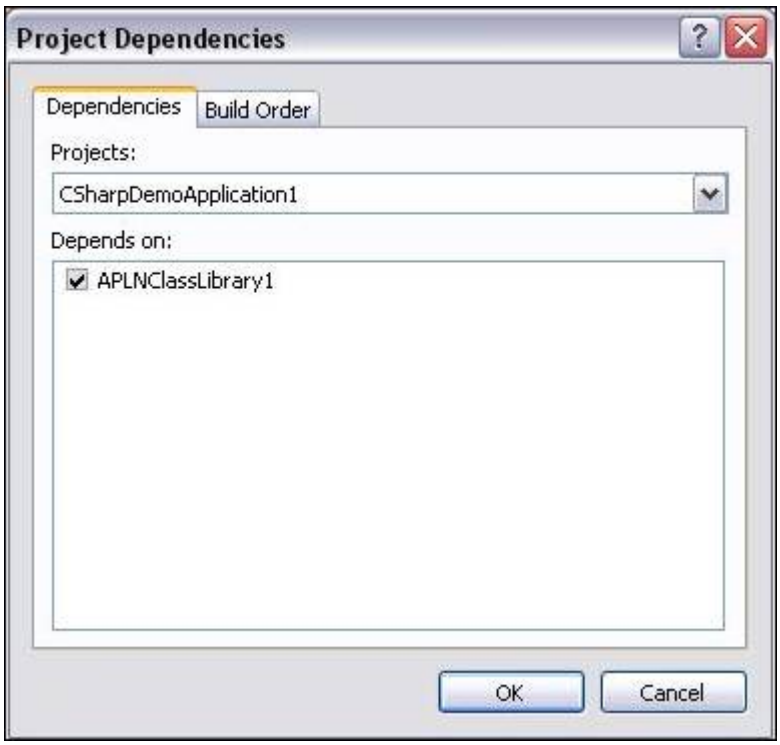


Figure 21. Setting project dependencies manually

VS.NET prevents you from creating circular references or modifying dependencies that resulted from adding project references. To verify the build order at any given time:

Click the read-only Build Order tab.

Note: In VS.NET 2005, the Project Dependencies context menu item in the Solution Explorer does not exist for web applications, instead, you need to select Websites > Project Dependencies.

2.1 - Embedding Files As Resources

Embedding files as resources allows you to embed any given file directly into your produced assembly. For instance to display a company logo on your Windows application, you could produce a regular Windows assembly and link to an external image that you send along with your application. You can also embed the image right into the assembly you produce. This enables you to avoid shipping the external image and, more importantly, prevents the possibility of these two files becoming separated.

To embed a file as a resource, it must first be included in your solution. You can then select the file in the Solution Explorer and change the Build Action property in the Properties window. The build action tells the compiler what to do with the specified file. If you select the Embedded Resource build action, the actual bytes of the file will be stored inside the produced assembly (regardless of whether this is an EXE or a DLL).

At runtime, you can then extract the bytes using the following code:

```
Assembly oAssembly =
System.Reflection.Assembly.GetExecutingAssembly();

Stream streamOfBytes =
oAssembly.GetManifestResourceStream( mylogo.bmp );          "           "
```

After this retrieves the bytes from the given embedded resource, you have to convert those bytes back into the original file type (using Image.FromStream(), for instance, to convert it back into a picture). Notice how this code is orthogonal to the file type being embedded as a resource. This enables you to embed any file type: sound and movie files, PDF files, or even another assembly.

2.2 - Changing the Font Size of IDE Windows for Demos

It is a common practice when demonstrating VS.NET or your code, to increase the font size of the text editor so that everyone in the audience can easily see the demonstration. The font size can be easily increased by a couple of methods:

> Select Tools > Options > Environment > Font and Colors > Size.

This works great, except that the text in the Output windows, Solution Explorer, Class view, Macro Explorer, or in the file tab titles can still be very hard to read.

Control the size of the text in these elements as follows:

> In the same settings window, the first drop-down list reads Show Settings For. Change it to read Dialogs and Tools Windows.
>
> Set the font and the size here in this window.

You control the format of the text elements of the majority of the IDE windows. The changes come into full effect after you restart the IDE.

To increase the font of the Output window:

> Change Show Settings For to Text Output Tools Windows.

To reset any of these settings to their default installation values:

> Click the Use Defaults button.

Note: This button applies only to the currently selected item in the Show Settings For drop-down list, so repeat this step for every setting that you want reset back to the default settings.

2.3 - Dragging Files to Obtain a Full Path

A useful feature of VS.NET is the ability to drag files from your Solution Explorer directly into your code. If you do this in a source code file, it will simply insert the full path to the selected file into your code.

2.4 - Moving Any Window Around

Every window in VS.NET is movable, resizable, and dockable: the Solution Explorer or Macro Explorer; the Properties, Task, and Output windows; and even your Toolbox, Server Explorer, and Find/Replace windows. To move any window in VS.NET:

> Drag the title bar to the desired position.

As you drag a window close to a dockable region (such as tabs or near another window frame), an outline appears, allowing you to preview the result before dropping the window.

To dock and undock windows:

> Double-click the title bar.

You can also move the order of tabs in your tab windows. This includes the files tabs at the top of your editor.

While the ability to control window positioning gives VS.NET enormous flexibility, the preview outlines are too confusing to make this an intuitive interface. If you have moved the windows positions and would like them reset, you can always reset all windows positions to their installation defaults:

> Select Tools > Options > Environment > General > Reset Windows Layout.

With VS.NET 2005, you can also reset the windows positions. To do this:

> Select Window > Reset Windows Layout.

One aspect of moving windows around is the ability to create a split screen. Use the following steps to split the editor into two vertical screens complete with their own set of file tabs (see Figure 23):

> Drag the tab of any open file and move it to the right of your editor (to the left of where the Solution Explorer usually resides).

This docks your selected file to the right and splits the editor into two vertical screens.

To close vertical split mode either:

> Close the second set by clicking the small X at the top right, or
>
> Drag the file tabs back to the left along with the other files.

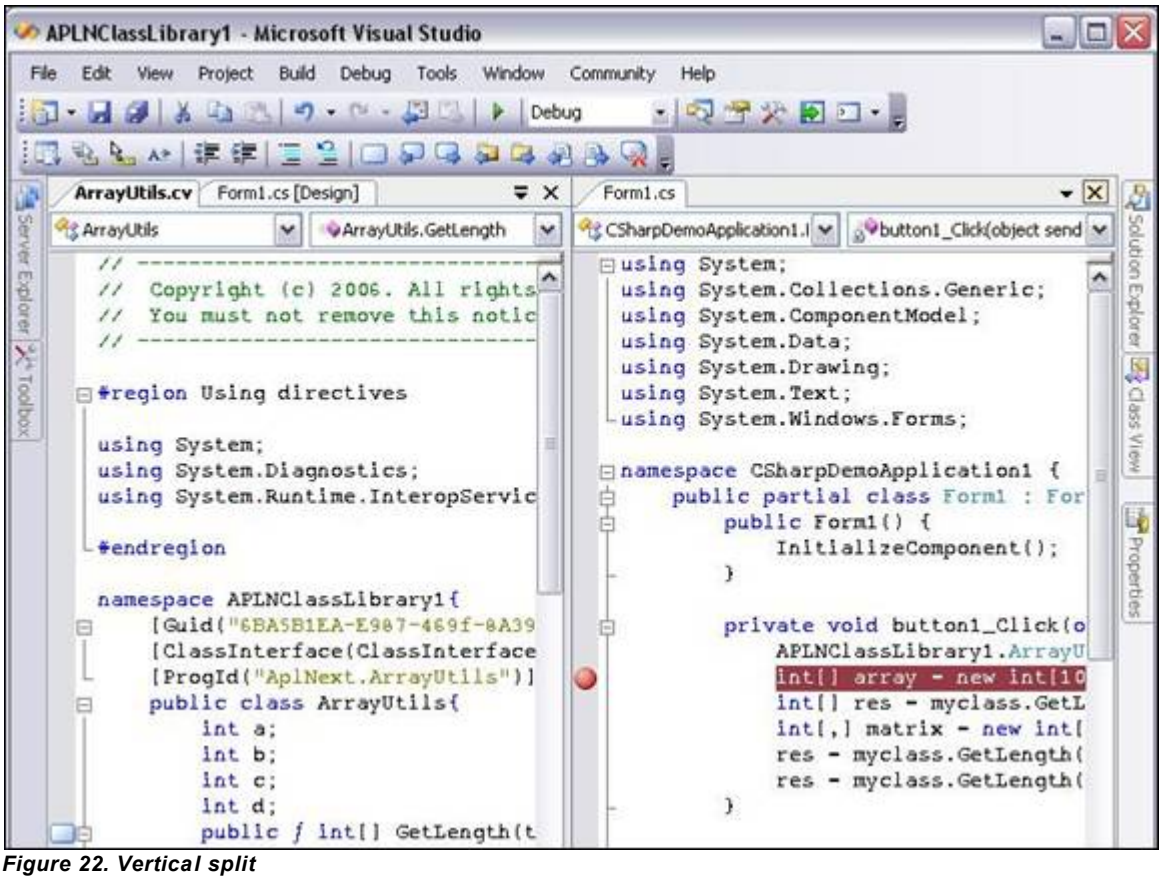


Figure 22. Vertical split

To create a horizontal split screen:

> Drag a file tab to the bottom of your editor.

2.5 - Creating Split Screens in the Same File

The "Moving Any Window Around" trick described in "Moving Any Window Around" shows how to create split screens so you can see two files next to each other. What if you want to create a split screen to see two locations of the same file? To do this:

select Window > Split

The horizontal divider can also be generated using a faster method:

Move your cursor right above the vertical scrollbar of the main editor. There is a very thin, short, rectangular-shaped divider (see Figure 23).

Place your mouse over that divider, the mouse icon changes to the divider icon.

Drag the divider down to the center of the screen to create the split screen (see Figure 24).

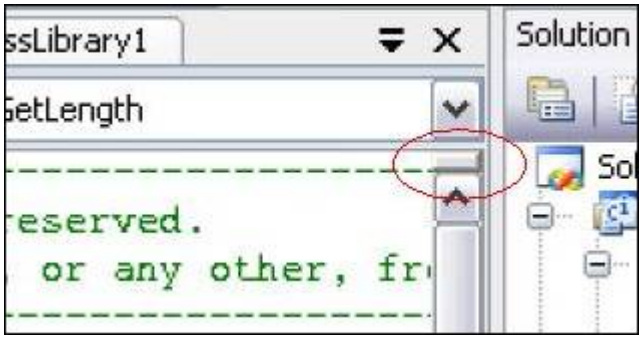


Figure 23. Horizontal split divider

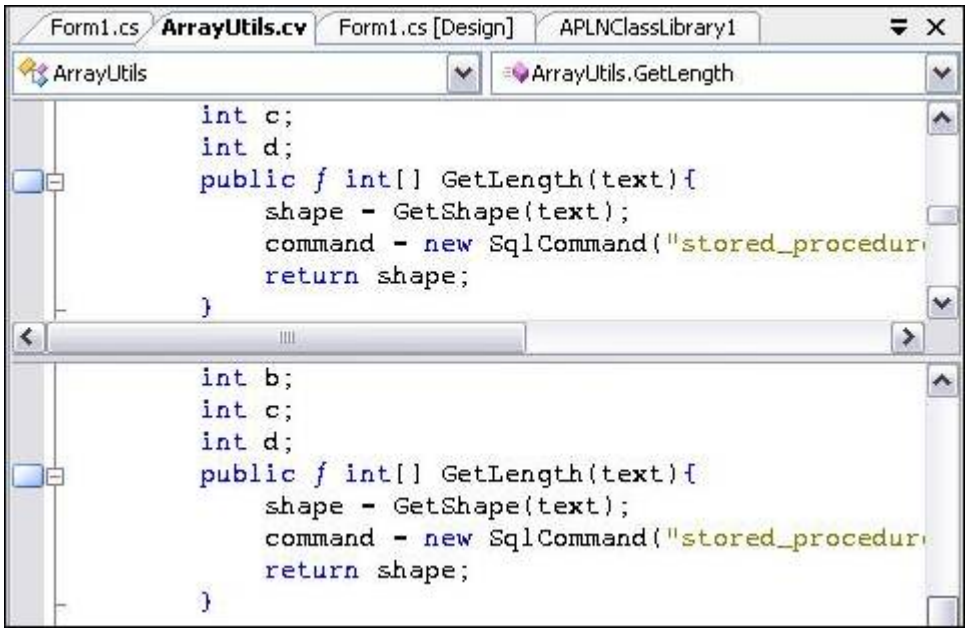


Figure 24. Split document.

To move the divider back to the top of your editor window:

Select the divider bar and slide back into its original position.

2.6 - Customizing the VS.NET Menu and Toolbars

The VS.NET menu can be customized in a variety of ways. You can add and remove commands as well as reorder them. To customize the menu and toolbars:

Select Tools > Customize.

With the Customize dialog box open, navigate back to the VS.NET menu.

The menu now does not react to left mouse-click events and will show context menus when you right-click the

menu items. Here you can rename, edit, and delete menu items; drag menu items around; or even create your own cascading menu groups.

You can also manage the icons for each menu item by right-clicking the item and selecting Choose Button Image from the pop-up menu. If you are not satisfied with the icons in the selection, you can copy icons from other menu items to your newly created menu ones. To copy icons from other menu items:

Right-click a menu item with the desired icon.

Choose Copy Button Image from the pop-up menu

Right-click the menu item you want to modify.

Choose Paste Button Image from the pop-up menu.

To add other commands to a menu:

Drag a command from the Command tab directly into the VS.NET menu.

Note: In addition to directly modifying the VS.NET menu items as long as the Customize dialog box is open, VS.NET 2005 adds a complete new GUI to modify the menu. The new GUI appears when you select Tools > Customize > Rearrange Commands. Here you can move, add, and delete menu items as well as toolbar buttons (see Figure 25).

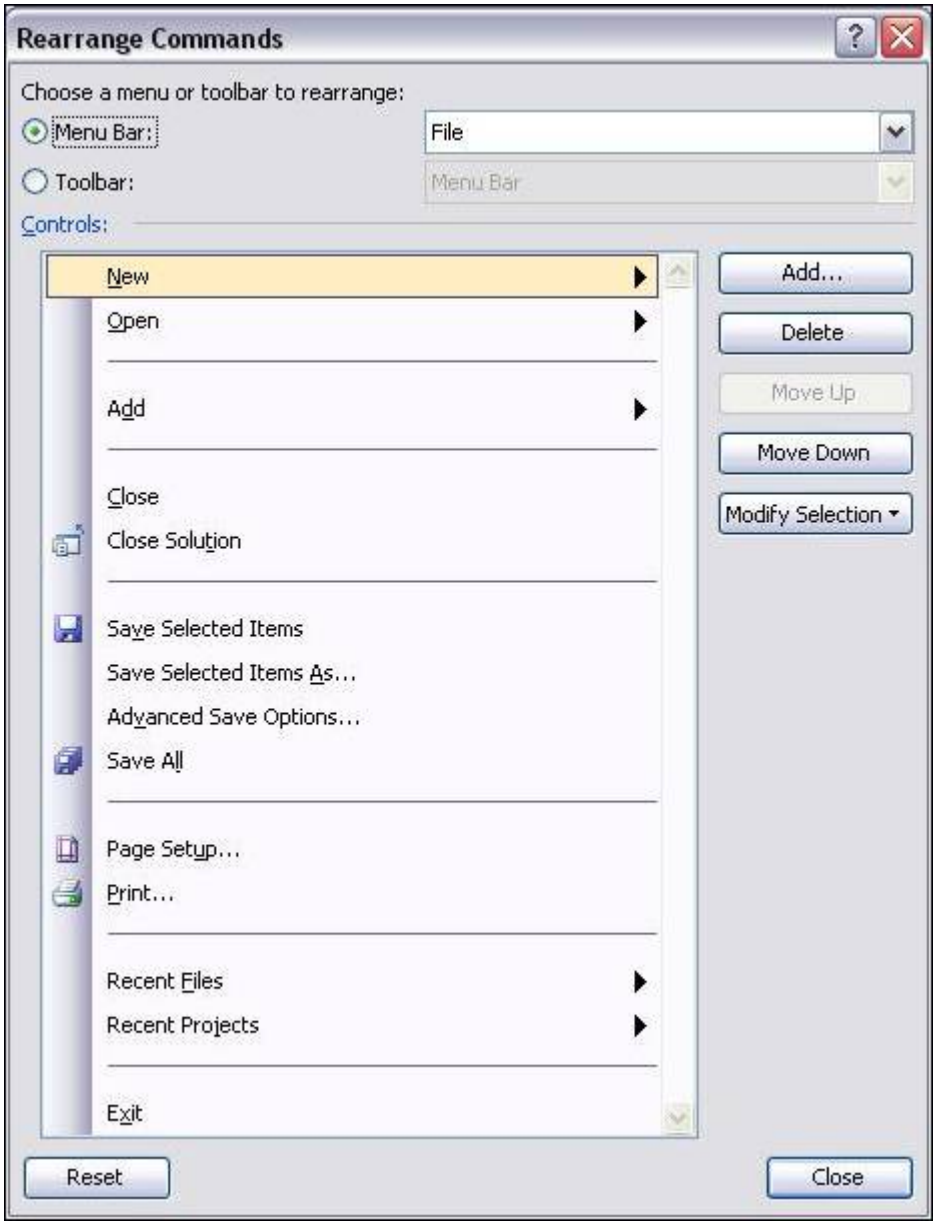


Figure 25. Customize menus using the rearrange commands

2.7 - Dragging Files from Windows Explorer into VS.NET

Visual Studio .NET completely supports file drag (and drop) actions. It allows you to drag files from Windows Explorer directly into VS.NET. If you drop them in the Solution Explorer under a project, it will first be copied into the same directory as the project and then included into the project. If you drag them into the code editor, VS.NET will either start the default external viewer (for example, Adobe Acrobat for PDF files) or display the file's contents inside VS.NET if it's a file type that it understands.

To drag files from Windows Explorer into VS.NET if you don't have enough screen space:

Drag the file into the Windows taskbar at the bottom of your screen

Pause for a few seconds over the taskbar for VS.NET. The pause brings VS.NET into focus.

Drop the file into the appropriate location.

2.8 - Using Full-Screen Mode (Ctrl – Shift – Enter)

Full-screen mode allows you to hide virtually everything except the main editor, where the entire screen shows the main view. To enter full-screen mode:

Select View > Full Screen or

Press Ctrl-Shift-Enter.

The main menu is still visible at the top, and a floating button that closes full-screen mode is also available. To hide the Close Full Screen mode button—you need to memorize the keyboard shortcut that returns to normal mode or:

Select View > Full Screen again.

Full-screen mode is available for any view, including the HTML, Designer, and XML views.

2.9 - Copying the Fully Qualified Name of a Class

The Class view is a hierarchical view of all your classes and namespaces in your solution. To display this view:

Select View > Class View or press Ctrl-Shift-C.

To go to any class and its members and navigate to the member definitions:

Double-click on the desired item.

Another useful feature allows you to extract the full namespace of any class or member:

Highlight the class or the class member.

Press Ctrl-C.

This copies the complete namespace of the selected item to the Clipboard. This feature comes in handy when you have a complex or deep namespace structure.

To paste the namespace into the VS.NET code editor, there is no need to copy it to the Clipboard first. Use the following method:

Drag a class or member of a class from the Class view directly into your code

Watch VS.NET paste the complete namespace and member name there.

2.10 - Changing Properties of Several Controls

When designing your Windows forms, you can use the Properties window to modify a control's behavior and appearance. The Properties window, however, is adaptable when you select several controls at the same time. To select a series of controls either:

Hold down Ctrl or Shift when selecting controls or

Draw a selection rectangle with your mouse,

The Properties window automatically displays the properties that are common to all of the selected controls. With all controls selected, any change you make in the Properties window affects all selected controls.

This is useful for instance, after you drag a series of text boxes from the Toolbox onto your form and want to get rid of the default "TextBox1," "TextBox2," etc. values.

Select all the text boxes.

Change the Text value to a single space by pressing Spacebar.

Change it back to an empty string by pressing Delete.

Do this twice because the initial values of each text box differ originally, so the Text property displays an empty string as the "common value".

This deletes the default text in all of them.

2.11 - Locking Controls

When laying out windows controls on Windows forms, you can easily move the controls around or create event handlers by simple dragging and double-clicking. However, this simplicity has its drawbacks as you can move things around accidentally very easily. This can cause problems if you have already finished designing your Windows forms. In order to prevent this from happening, you can lock your form.

To lock the position of your controls on your form:

While in the Designer view, right-click anywhere on your form

Choose Lock Controls from the pop-up menu (see Figure 26).

You still have the ability to add event handlers and modify a control's appearance, but you can no longer accidentally move or resize a control. To indicate that it is locked and unmovable, a thin, black outline appears around each selected control.

To return to the Designer view:

Right-click your form and choose Lock Controls from the pop-up menu again.

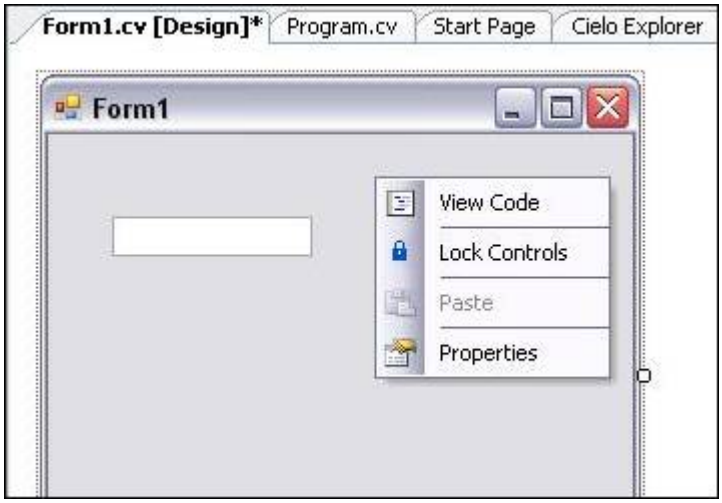


Figure 26. Lock controls in a form

2.12 - Toggling the Description in the Properties Window

The Properties window not only displays all properties of a selected control, but the Description pane at the bottom briefly describes the active property. As you select different properties, the Description box informs you what the selected property does. To turn off the Description box panel:

Right-click the property name.

Choose Description from the pop-up menu.

To turn it back on use the same method.

2.13 - Change Drop-Down List Values in the Properties Window

Whenever a property only accepts a finite set of values, the value field becomes a drop-down list, from which you make your selection. For instance, the FormBorderStyle property of a Windows form only accepts None,

FixedSingle, Fixed3D, FixedDialog, Sizable, FixedToolWindow, and SizableToolWindow. To select the appropriate item:

Open the drop-down list.

Select the style you want.

Anytime you have a drop-down list in the Properties window, you can iterate over the list more quickly by simply double-clicking the property or its corresponding drop-down list. Without expanding the list first, double-clicking it sets the value to the next available item in the list (or to the first item if the current value is the last one).

This trick can be extremely useful when switching Boolean values because a double-click changes the value quickly from True to False, or vice versa.

2.14 - Adding and Removing Event Handlers Through the IDE

Adding default handlers through the IDE is quite easy. In most cases, you only need to double-click a control which creates the necessary code for the default event handler.

Adding and removing non-default events handlers is still easy, but, in Visual APL, it requires not only the removal of the method itself but the removal of the code that hooks an event handler to an event, often found in the InitializeComponents() method.

The proper, but relatively hidden, way to add and remove event handlers in Visual APL is to use the Properties window:

Select the control

Click the Events button in the Properties window (the yellow thunderbolt).

The Property window displays all the events that the selected control exposes, along with any event handler that is already hooked up to them.

In addition, the event handler fields are clickable (see Figure 27).

To create an event handler:

Double-click an empty field.

Choose which event you want to subscribe to.

To hook an event handler which is already written, to an event:

Use the drop-down button next to the selected field that automatically lists all matching event handlers.

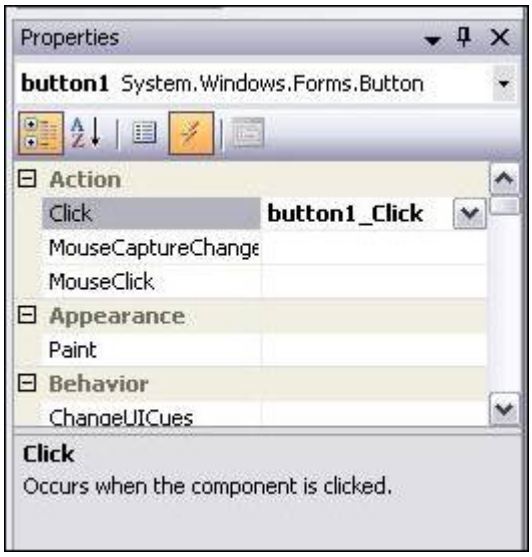


Figure 27. Setting events

To delete an event handler:

Delete the value in the event field.

This also removes the event handler subscription you have in the InitializeComponents() method.

2.15 - Selecting Control Through a Drop-Down List

When there are many controls on a Windows form, it can become a challenge to find a specific control, and select it. This problem often occurs when many panels overlap one other or when the Windows form becomes too crowded to isolate a specific control that you want to modify.

To select a specific control:

Select the drop-down list that appears right above the Properties window.

Select the desired control.

Note: This drop-down list is only populated in the Designer view. It contains all the controls that exist on the Windows form. To select a certain control, you just need to know its ID and data type.

Chapter 3: Compiling, Debugging, and Deploying

Not only is VS.NET a great editor, it is also a powerful compiler, debugger, and profiler. It allows you to precisely control your compilation procedure and provides the features which are absolutely essential in to locating and fixing a bug: analyzing your code, attaching to running processes that you want to debug, and changing code and variables at runtime. This chapter covers topics that you need to know when it comes to compiling and debugging your programs.

3.0 - Setting the Default Namespace and Assembly Name

Following the official naming guidelines suggested throughout the industry, you would declare your classes in your own company and project-specific namespace. Typically, you end up with the following namespace hierarchy (at a minimum):

```
MyCompanyName.MyProject.MyClass
```

When you add new classes with the Add New Item dialog box, VS.NET does not place your new class in any project namespace. It places it, by default, in the top-level namespace, which usually means the name of your assembly. To set the default namespace when you create new projects:

Select Project > Properties > Application.

Specify the default namespace in the Default Namespace field.

This namespace can be many levels deep; new classes added through the VS.NET dialog box will be placed in that specified namespace. In addition, you can also control the name of the assembly that is being generated by specifying it in the Assembly Name field. While Windows applications typically use one word for the assembly name, Control Library projects should be named using the same guidelines as the namespace.

3.1 - Generating Compiler Warnings Through the Obsolete Attribute

A commonly used way to display warnings in VS.NET at compile-time is to set an Obsolete Attribute to a method. Throughout the product development cycle, occasionally certain methods become obsolete. Sometimes the old method is not useful anymore. It may have become inefficient, or has been replaced by another method. If you can't modify those methods, will need to write another implementation of the method using a slightly different name or signature. To maintain compatibility, you do not want to remove the old method and break your code. This is where the Obsolete attribute comes in handy:

```
[Obsolete( Use the new MyMethodEx"instead  )]            !       "
public void MyMethod()...
```

Setting the Obsolete attribute as above makes a warning message appear in the Task List stating that the particular call to a method is obsolete. The warning message also includes your personalized message that you pass as the attribute's argument (such as, "Use the new MyMethodEx instead!").

As with the warning compiler directives, this method does not affect the compilation behavior in any way. You also must activate the Task List to see these warnings. Unlike warning compiler directives, the warning only appears if there is code that tries to invoke the obsolete method. These warnings will never appear if you don't refer to these methods anywhere in your code.

3.2 - Setting the Assembly Output Path

When you build a project, the produced assemblies are typically placed in the \bin\Configuration subfolder of your project folder, where the configuration folder is typically Debug or Release.
These are the default settings. To specify another directory where you want to place the produced assemblies and external files:

Select Project > Properties > Build for Visual APL projects.

Place either a relative or absolute path in the Output Path field.

This setting is used at the next build.

These configuration-specific properties allow you to specify a different output path for each configuration. For instance, if you want, you can set the default output path for the Debug release as the usual bin subfolder, while directing the release build directly to a network share on your internal company network.

3.3 - Setting the .NET Framework Version for Your Assembly

A great side-by-side installation feature of the .NET Framework is the ability to have multiple versions of the .NET Framework installed on a given computer, without any of them interfering. By default, all non-web applications use the .NET Framework with which they were compiled (if available), whereas web applications by default always use the most recent version of the .NET Framework.

You can specify which .NET Framework is supported and required for your assembly by modifying the application configuration file (MyApplication.exe.config or Web.config). What you need to do is:

Insert the appropriate Configuration/startup/supportedRuntime and Configuration/startup/requiredRuntime XML tags in the configuration file

Set its version attribute to the specific .NET Framework version.

This enables you to force a Windows application to use an older version of the .NET Framework.

This configuration modification is easy in VS.NET. To set this for Visual APL:

Select Project > Properties > General > Target Platform.

Set the supported and required runtime versions for your assembly (see Figure 28).

To verify that your assembly is picking up the correct version, check the Version property in the .NET Framework class System.Environment.

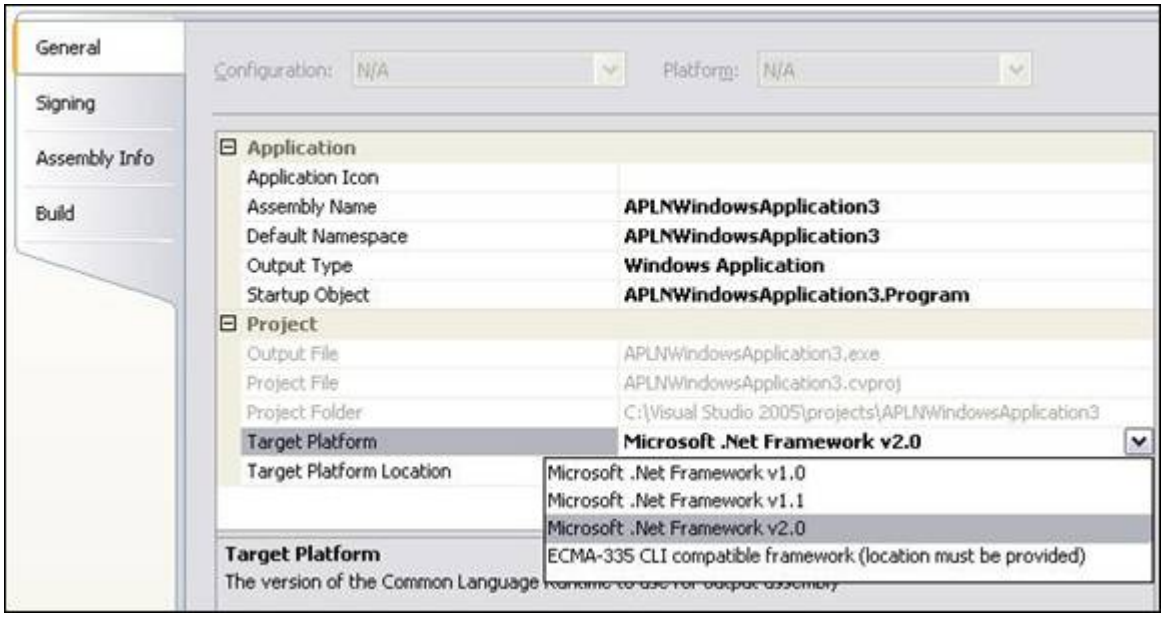


Figure 28. Choosing the target runtime.

Note: Supporting the 1.0 Framework, or even version 1.1 is an unsupported environment. Simple programs most likely will work, but for more complex programs you are strongly advised to check the compatibilities manually in case your code uses version 2.0–specific features.

3.4 - Moving the Next Statement During Debugging

When stepping through your program one line at a time, you may need to jump a few lines back. To do this:

Right-click an arbitrary line

Choose Set Next Statement from the pop-up menu (see Figure 29).

This forces the debugger to jump to that line and continue debugging "normally" from there.

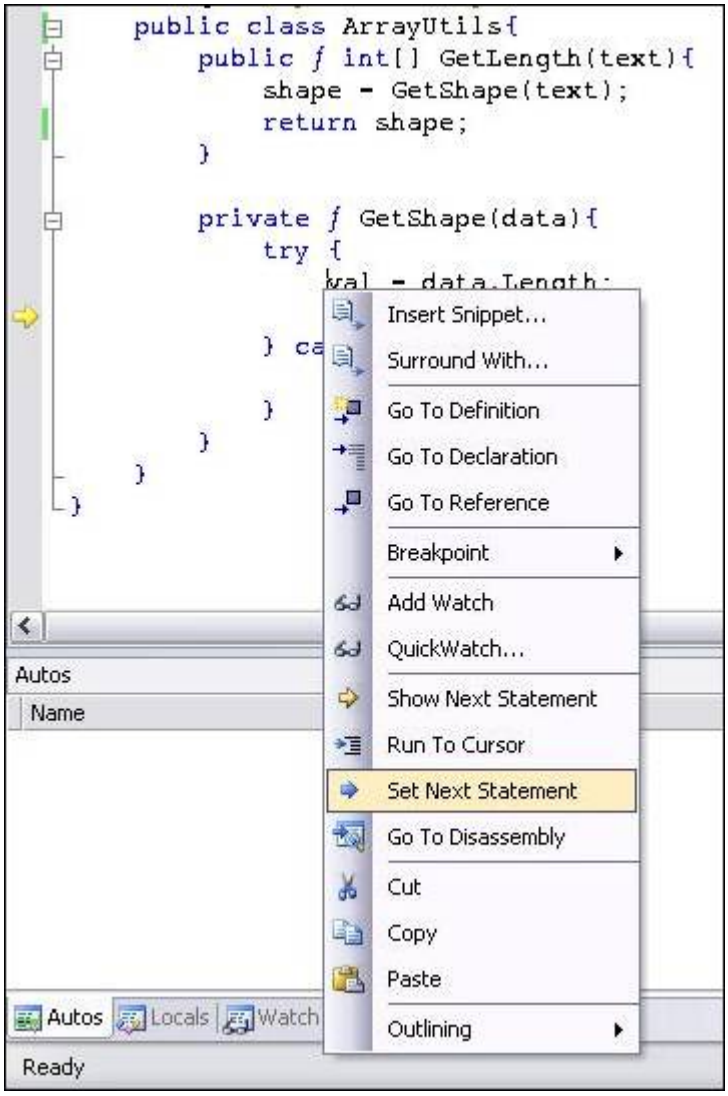


Figure 29. Set Next Statement

To jump back, and also jump forward in and out of control statements:

Drag the yellow arrow to any line.

Note: You cannot jump out of the current stack frame, so you are limited to moving inside your current method.

In addition, moving the current execution line can bring your program into states that under normal execution could not occur. Still, it's an extremely useful feature to rerun certain code lines without restarting your debugging session.

3.5 - Changing Variable Values in the Watch Window

In addition to moving the next-statement pointer, you can change variable values at debug-time. In the process of debugging your application, you may have moved your variables of interest into the Watch window

(probably by dragging your variable there). The Watch window does more than display the current variable value and type; the value field is also editable.

For most value types this is accomplished by entering the new value.

Note: You need to change the internal tick value of DateTime variables.

As for reference types, you can re-reference variables to other variables. Let's say you have two instances of hash tables in your Watch window, named foo and bar. Setting the variable foo to the reference bar's hash table is as easy as typing bar in foo's value field. You can only change a reference variable to another reference variable of the same type (or its derived types).

Note: This can bring your program into states that under normal conditions would never be encountered.

3.6 - Executing SQL Procedures Through the Server Explorer

The SQL Server tree branch in the Server Explorer allows you inspect and analyze a SQL Server instance. In addition to the general features of inspecting a database table and Excel-like modifications of table contents by editing rows, the Server Explorer has other useful features.

VS.NET has limited capabilities of editing stored procedures. To view, edit, and modify stored procedures:

Right-click any stored procedure.

Choose Edit Stored Procedure from the pop-up menu.

Unfortunately, this feature does not compete well with the Enterprise Manager because error messages regarding syntax error are too general. Nevertheless, it's quite useful for its designed purpose of viewing, editing, and modifying stored procedures.

To execute stored procedures at design-time:

Right-click a stored procedure.

Choose Run Stored Procedure from the pop-up menu.

VS.NET inspects your stored procedure's parameter list. If necessary, the Run Stored Procedure dialog box is displayed:

Enter each parameter's value.

Execute your stored procedure and see the results.

3.7 - Customizing the Call Stack

A stack trace is a visual representation of the current hierarchy of method invocations as VS.NET steps through your program. While debugging your program, you step into methods and methods within methods. The stack trace keeps track of all these different levels.

To see the current stack trace:

Select Debug > Windows > Call Stack or

Press Ctrl-Alt-C,

Each method invocation is displayed on its own line, including the line-number and argument values. Each new method invocation is known as a stack frame.

The stack trace has been around in Visual Studio for a long time and is a widely known tool. The advantage of the stack trace window is that it allows you to identify how you get to the current execution point and also inspect the arguments that have been passed to the methods.

To make VS.NET immediately jump to the method invocation on a particular level of your program:

Double-click any line in the stack trace.

A relatively unknown aspect of the stack trace is that you can customize the Call Stack window. To do this:

Right-click the call stack.

Customize what appears there (see Figure 30) according to your requirements.

In addition, you can send the information regarding a single method invocation to a coworker:

Copy a stack frame to the Clipboard by pressing Ctrl-C.

To send your coworker the entire call stack:

Press Ctrl-A first, or

Before copying the selection to the Clipboard, Choose Select All from the context menu that appears after you right-click.

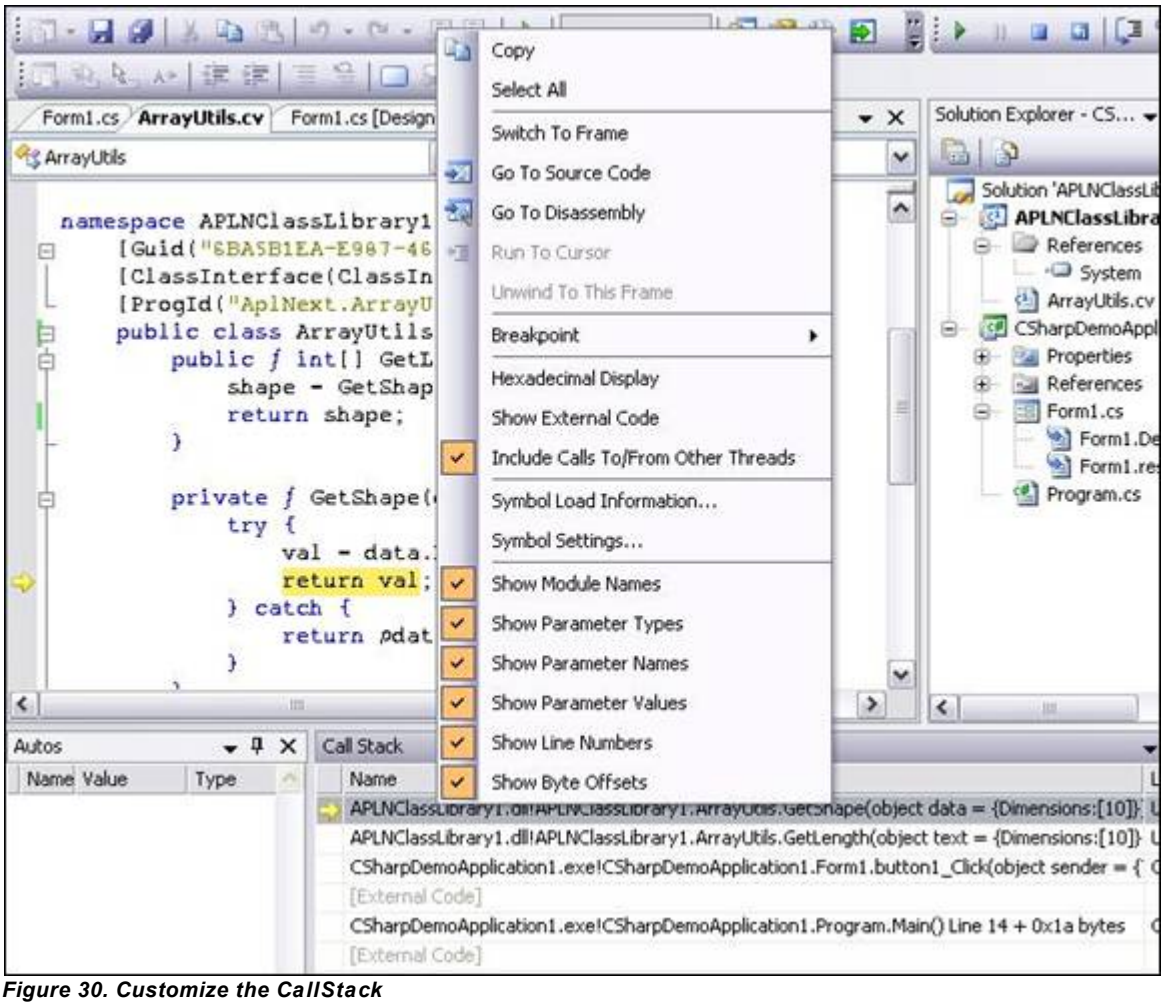


Figure 30. Customize the CallStack

3.8 - Attaching VS.NET to an Already Running Process

To instruct VS.NET to debug your program, you first are telling it to build your project (if necessary) then start the program in debug mode. This means that VS.NET is attached to the program so that it can react to breakpoints and other debug-related methods, assuming that the project was built with the debug release. To debug your program:

Press F5

There some cases, where you need, or want, to debug an already running process that has not been started with VS.NET you must:

Open the project for the program that is already running.

Select Debug > Attach to Process

A list of all active processes on your machine is displayed.

From the Processes dialog box, select the process you are interested in debugging and click Attach.

3.9 - Debugging Several Projects Inside the Solution

In a multi-project solution, VS.NET will start the project that you have marked as the "startup project." That project is indicated in the Solution Explorer with bold letters. If you start the other projects through Windows Explorer, you will see that VS.NET does not hit any breakpoints for those projects because VS.NET was not attached as a debugger to them.

It is possible to debug those programs anyway, using the instructions in, "Attaching VS.NET to an Already Running Process."

To instruct VS.NET to start a project and attachs itself to a specific program:

Right-click your project

Select Debug > Start New Instance from the pop-up menu.

You can repeat these steps several times to start multiple instances of your program and still debug them all. This is useful in debugging multi-threaded client-server scenarios.

Tell VS.NET which projects you want to start on each new debug session (see Figure 31):

Right-click your solution

Choose Set Startup Projects from the pop-up menu.

By default, VS.NET uses the Single Startup project, where only one project is started.

To start more than one project:

Switch to Multiple Startup Projects

Modify the Action value for each property: None, Start, or Start Without Debugging.

To control the order by which these multiple projects start:

Click the Move Up or Move Down button to position your projects in the list.

In a client-server scenario, you can use this to make sure that the server program is started before the client program.

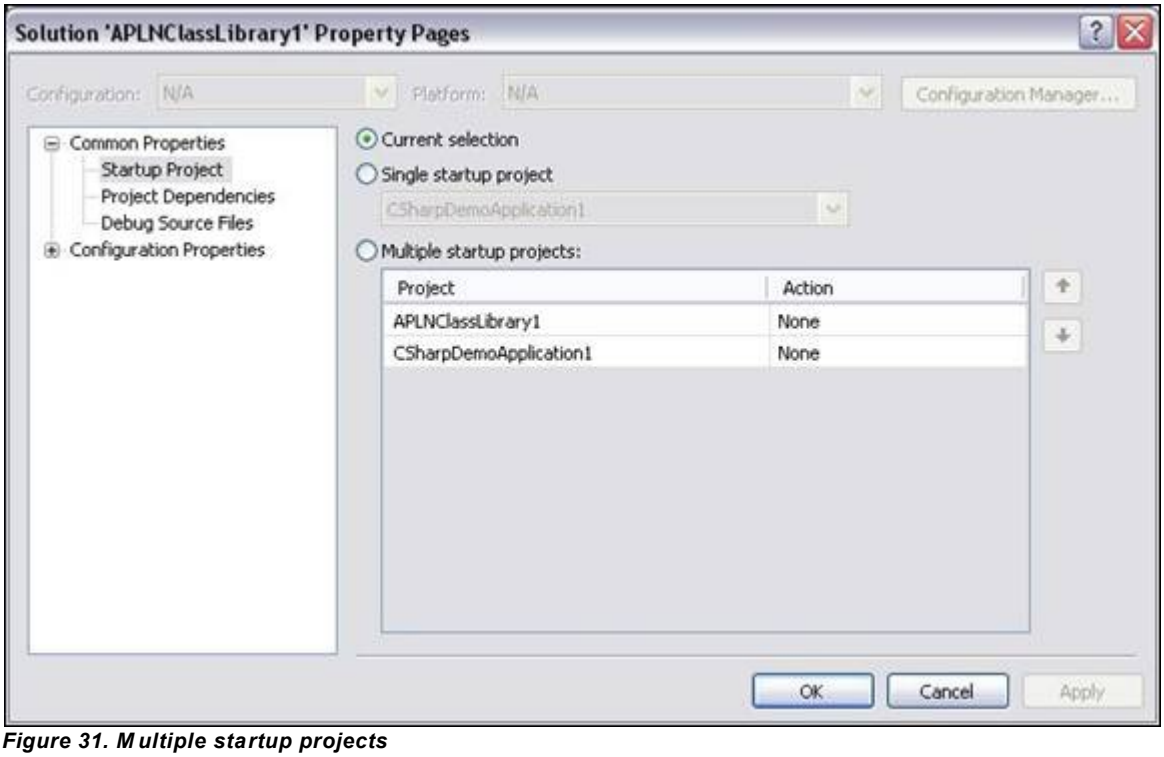


Figure 31. Multiple startup projects

3.10 - Breaking Only for Certain Exception Types

A good program usually catches all possible exceptions that can be thrown at runtime. However, this makes it a bit difficult for developers to debug a complex program that is still in development. Because there aren't any unhandled exceptions, VS.NET never catches an exception or prompts the user to break into the code whenever a specific exception is being thrown.

To specify the exceptions that developers are interested in defining, there is a setting in VS.NET. To utilize this setting:

Select Debug > Exceptions, or

Ctrl-Alt-E.

A tree view–style list of all possible exceptions that VS.NET can hook into (see Figure 32) will be displayed.

In addition to the many Common Language Runtime exceptions, you can hook into C++, Native Run-Time checks, and Win32 exceptions.

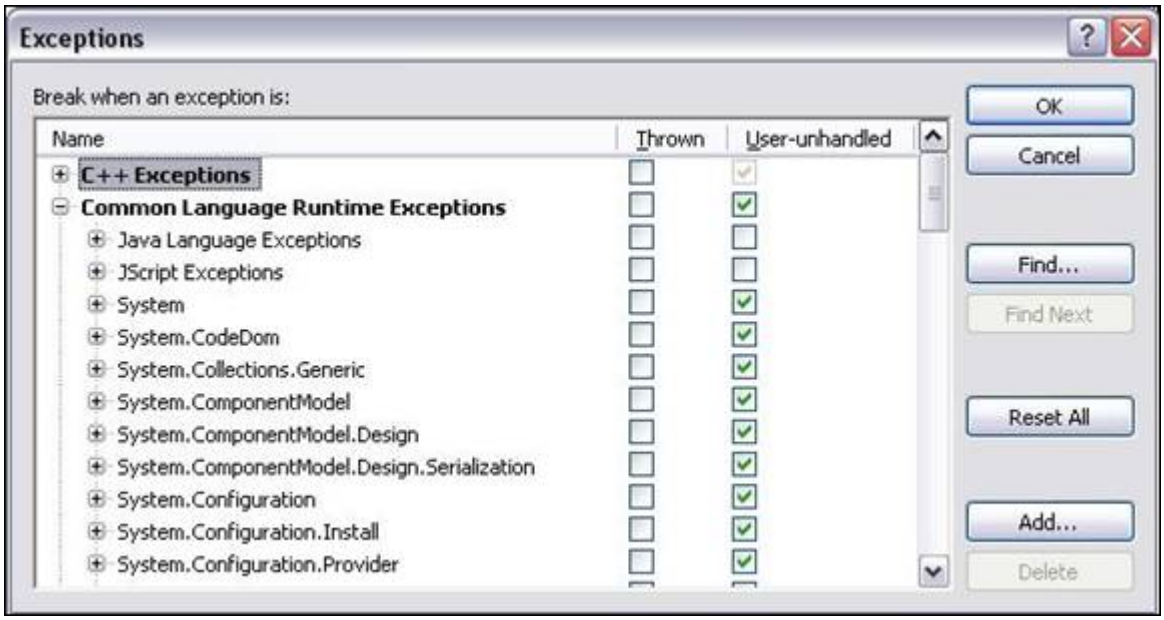


Figure 32. Break on specific exceptions

From this list you are able to:

Set, for each possible exception, exactly when to break into the debugger.

You can either hook into the debugger when a specific exception is thrown or when an exception is not handled. In the predefined .NET exceptions, you can hook into your own .NET exceptions.

To specify the complete, fully qualified string that defines your .NET exception, for example, "MyCompany.MyProduct.MyBusinessException":

Click the Add button in the Exceptions dialog box.

3.11 - Breaking Only When Certain Conditions Apply (Ctrl – Alt – B)

A heavily used method to add or remove exceptions is by clicking the gray vertical bar to the left of the editor. Clicking it adds and removes the red circle that indicates a breakpoint. By doing so, many developers never encounter the very useful conditions that you can set for breakpoints.
To access these conditions:

Set your breakpoint using your normal method.

Right-click your breakpoint.

From the context menu, choose Condition (Figure 33) to get to the Breakpoints window (Figure 34).

Two buttons stand out at the bottom of the Breakpoints window. To specify a condition under which a breakpoint becomes active:

Enter a .NET expression.

This can either be simply a variable name ("myBoolVariable") or a more complex .NET expression ("((System.DateTime.Now.Second % 10) == 0)"). You can choose to break into the debugger if the expression evaluates to True or when the expression value changes. Naturally, for the first option, the

expression has to evaluate to a Boolean value. For the second option, your expression can be anything. VS.NET breaks into the debugger only if the runtime value of that expression changes from the last time it passes by this conditional exception (this implies that program execution has to pass by this code segment at least once previously, before it can recognize a change in value).

Given the flexibility of the expression, this feature can be very powerful. For instance, you can debug a snapshot of a DataSet only if the DataTable row size is greater than 0.

In VS.NET 2005, all the above-mentioned conditions are accessed in the following way:

Sset your breakpoint as you would normally do.

Right-click your breakpoint.

From the context menu, choose Condition to get to the same screen (see Figure 62).

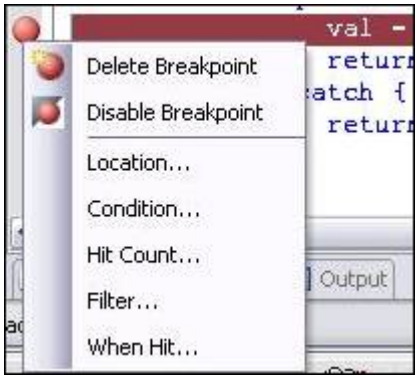


Figure 33. Set breakpoint condition.

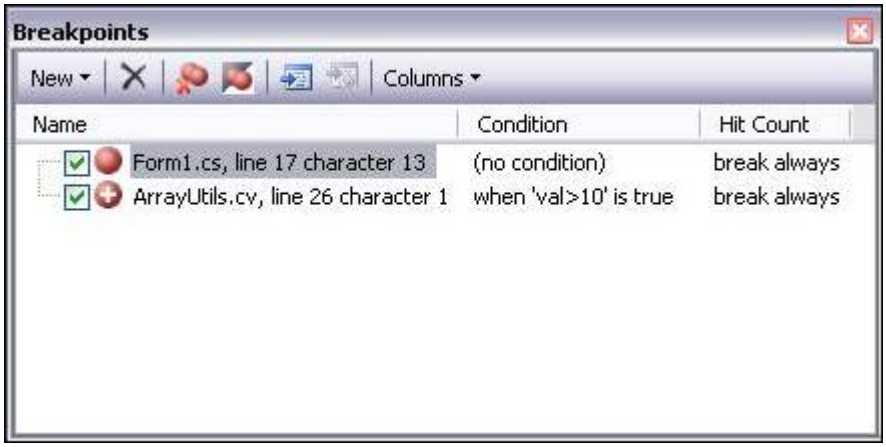


Figure 34. Breakpoints Window

To see and modify the condition in the Breakpoints window:

Open that window by selecting Debug > Windows > Breakpoints or

Pressing Ctrl-Alt-B.

A list of all breakpoints that you have set, along with their conditions will be displayed.

Note: You can disable breakpoints from this window as well, using the check boxes, or jump to their location in the file by double-clicking them.

3.12 - Saving Any Output Window

The Output window (Ctrl-Alt-O) shows a lot of trace information regarding your program execution. It lists whenever the .NET Framework loads a DLL for your application and, probably more importantly, all the messages that you have emitted with System.Debug.WriteLine.

To save all these trace logs:

Press Ctrl-S to save the entire output to a file.

To search through the Output window:

Press Ctrl-F

You can even apply some of the other editor tips and tricks such as Ctrl-C for copying an entire line or Ctrl-R, Ctrl-R for word-wrapping (although VS.NET 2005 now offers a button for word-wrapping in the Output window).

3.13 - Aligning UI Elements Automatically

If you are positioning UI elements in a Windows form, you have probably noticed various colored lines that appear on the form as you move or resize elements (see Figure 35). This allows you to snap your UI element to vertical or horizontal lines. Solid blue indicates lines to which other UI elements have already been snapped; they help you align elements consistently. Green dotted lines indicate the default margin between the UI element you are moving or resizing and the elements around it; they help you maintain uniform spacing between elements. Finally, solid red lines indicate that the text inside the current element is aligned with an adjacent UI element or its text.

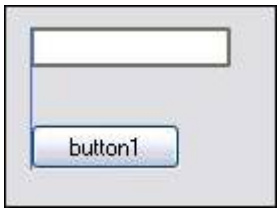


Figure 35. Align lines

To position UI elements without snapping to these colored lines:

Press Alt to turn off automatic alignment temporarily.

To switch back to the grid where all UI elements are aligned to a predefined grid:

Select Tools > Options > Windows Forms Designer > General and change LayoutMode back to SnapToGrid.

Note: After changing that value, you need to close and reopen the Designer view to use the newly selected layout mode. In SnapToGrid mode, you can press the Ctrl key to move elements without snapping them to the grid.

3.14 - Adding a Standard Menu Strip

Standard Windows applications use a common set of top-level menu items. In most cases, they are File, Edit, Tools, and Help. VS.NET 2005 allows you to add these default menu items to your own Windows forms applications. To add your own menu strip:

Drag a MenuStrip to your Windows form.

With the MenuStrip selected, the description panel below the Properties window shows an Insert Standard Items link (see Figure 74):

Click that link.

VS.NET inserts these standard items onto your MenuStrip. Menu items you insert contain the default submenu items as well. For instance, the File menu includes the usual New, Open, Save, Save As, Print, Print Preview, and Exit items, along with its default shortcuts, hot keys, and icons.

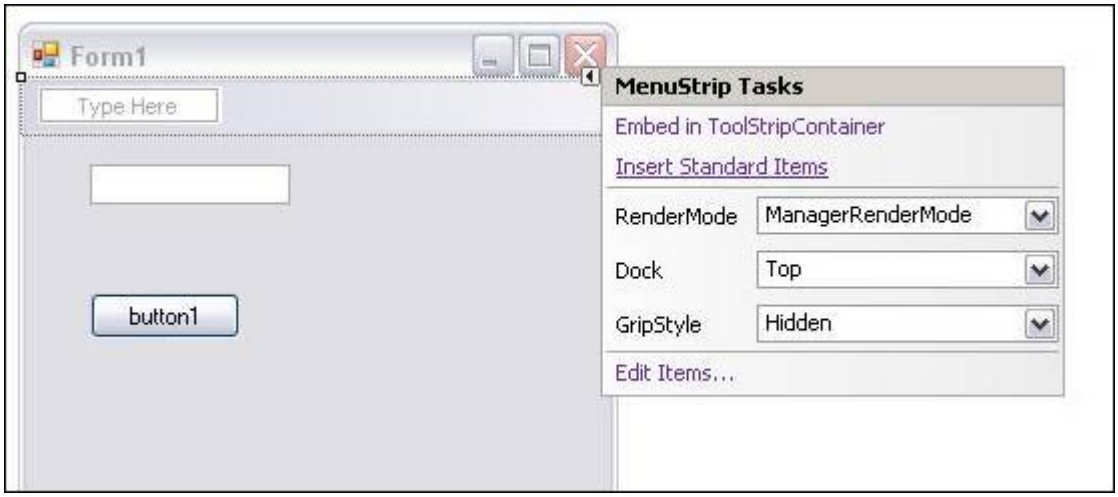


Figure 36. Menu strip standard items

3.15 - Setting the Tab Order of Controls

The tab order is the order by which controls on the form receive focus as you press the Tab key. You can control this order by setting the Tab Index property of each control to a number that corresponds to the position in this order. This can prove difficult at times because you don't know—and can't see—the other controls' tab index unless you select them.

Figure 37 - Tab Order button on the left of the Layout bar

VS.NET 2005 introduces a new way to set the tab order: the Tab Order button on the Layout bar (see Figure 37).

Click the Tab Order button to display the tab index for all UI elements on the form.

You now see all the tab indices.

Click repeatedly on each UI element to set the tab order in linear fashion.

The first element you select is given a tab index of zero. The next one you select has a tab index of one, and so on. As you set the index for each control, the background color of the tab index caption switches from blue to white, so you can keep track of which UI elements you have already tagged. To prevent you from accidentally selecting a wrong UI element, a gray rectangle surrounds the element you mouse over for better identification.

When you are done setting the tab order:

Click the Tab Order button again or

Press the Escape key.

3.16 - Importing and Exporting IDE Settings

VS.NET is an extremely powerful tool with many things in the IDE that you can customize to suit your specifications. Because you will become accustomed to your particular settings, moving from one machine to another can cause problems if you are not able to move your IDE settings along with you.

VS.NET 2005 allows you to export your IDE settings to an XML file (the extension is actually ".vssettings"). To import it into another instance of VS.NET on another computer:

Select Tools > Import > Export Settings.

In the tree view shown in the Import/Export Settings dialog box, you are presented with all the customizable options you can export (see Figure 38).

Check the options you want to be part of your profile.

Export them to the .vssettings file.

Figure 38. Export VS Settings

Import the .vssettings file to another VS.NET IDE.

Select which settings you want to import and which ones to ignore.

In the same dialog box you can also reset your complete VS.NET IDE to a particular profile. These might be custom profiles that you saved before. You can also reset to the default installation settings (which is just another regular .vssettings file).

To create a master .vssettings file for coordination between co-workers, e-mail it to all your team members so that they can import it individually. You can also create the single .vssettings file and place it on a well-known network share on the intranet. To obtain these settings have the members of your team:

Select Tools > Options > Environment > Import > Export Settings > Team Settings.

There they have to turn on Track Team Settings File and point it to that shared .vssettings file. Next time they start their IDE, it will detect the file and import it. One advantage of this feature is that another trusted team lead can export a version of the shared .vssettings file and overwrite it, so that the IDEs of each developer will detect that change and import it upon the next startup.

3.17 - Closing All Other Windows

It's very common to have a lot of files open at the same time when developing your program. After working for a while, you might have several dozen files open and want to close all of them except the one on which you are currently working.

To close all the open files:

Right-click one of the file tabs.

Choose Close All But This from the pop-up menu.

This option does exactly what it says (see Figure 39).

Other menu options new to VS.NET 2005:

1. Open Containing Folder
   -starts up Windows Explorer and opens the folder in which your file is located.

2. Copy Full Path
   -copies the full file path of the selected file into the Clipboard.

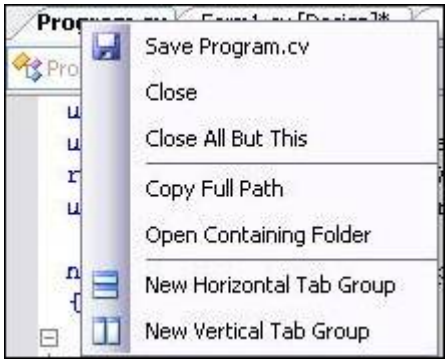


Figure 39. File tabs options

3.18 - Showing Shortcuts for All Buttons

Using and memorizing shortcuts whenever available, gives you a strong advantage when developing. It naturally increases your speed and therefore your efficiency. Keyboard shortcuts prove to be faster than manipulating the mouse. Many VS.NET menu and submenu items have these shortcuts which are seen every time you click the menu item.

This reminder is also available for the toolbar buttons. To see this reminder:

Select Tools > Customize

Check both the Show ScreenTips on Toolbars and Show Shortcut Keys in ScreenTips options.

Now as you mouse over a button, the ToolTip that appears after a small delay will also show the button's keyboard shortcut, if available.

Introduction

Visual Studio .NET comes complete with many features and functions that dramatically increase our efficiency as developers. As a powerful code editor, compiler, and debugger, it contains features to stress-test, analyze, and optimize your code, and allows easy integration with code documentation, reporting, or smart-device programming, such as the Pocket PC.

Because of the sheer number of features that Visual Studio .NET contains, it is a challenge for .NET developers to become familiar with all of its features, shortcuts, and functionalities. The beginner developer will find a virtual treasure trove of features with which to start, while advanced Visual Studio .NET users will appreciate the many new features and improvements the new Visual Studio .NET 2005 brings.

Most of these features and functionalities are documented, and are accessible through the VS.NET main menu or context menus. But, because of the vast number of features with which VS.NET is equipped, developers don't always know them or use them. This guide should help familiarize developers with the tips and tricks that are at their disposal in this powerful tool and their specific application to the Visual APL language.

Chapter 1: Editing Code

While you create code, there are many techniques and shortcuts that allow you to write and navigate through your code quickly and easily. This chapter introduces many of the tips and tricks you will need for such tasks as code navigation, performing complex find-and-replace searches, and generating code.

1.0 - Inserting Comment Tokens (Ctrl-K,Ctrl-H)

This feature enables you to write a comment and find it again easily. To see a list of all reminders you placed in your code (see Figure 1):

Without recompiling, select View > Show Tasks > All

To insert task shortcuts in your code:

Press Ctrl-K, Ctrl-H.

This marks the current line with a shortcut icon and inserts a clickable shortcut icon in the Task List.

To remove the shortcut:

Press Ctrl-K, Ctrl-H.

These shortcuts survive IDE restarts.

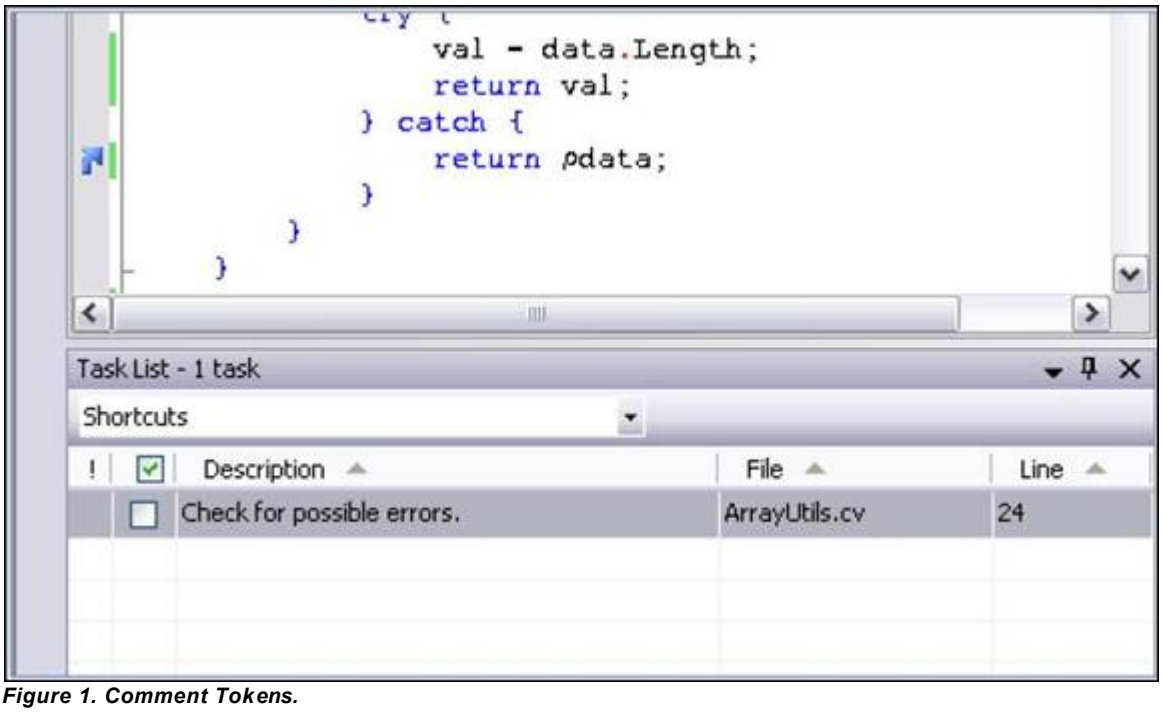


Figure 1. Comment Tokens.

1.1 - Commenting Code Blocks (Ctrl-K, Ctrl-C)

One-line comments are extremely useful in explaining seldom used code and assisting in navigation and definition of development projects. To inserted a comment for a code block or segment:

Press the "//" token for Visual APL.

Additionally, Visual APL allows you to comment entire paragraphs and segments. To place a comment in a paragraph or segment:

Select "/#" (and corresponding "#/") tag around the comment.

To quickly comment entire paragraphs:

Select the text.

Click the Comment button (see Figure 2) or

Press Ctrl-K, Ctrl-C.

This comments an entire selection.

To uncomment any selection:

Click the Uncomment button or

Press Ctrl-K, Ctrl-U.

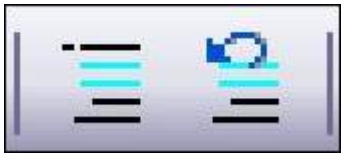


Figure 5 - Comment and Uncomment buttons

1.2 -Creating Regions

The more code you generate, the more difficult it can become to navigate. In addition to selecting classes and their methods from the drop-down lists above the main editor, you can also group your code into logical regions. Regions are extremely helpful for dividing code in logical ways and even commenting it. Regions allow you to collapse code to a single line defining the region and still easily see what is inside it once it is collapsed. They can even be nested. Automatically generated code in VS.NET usually uses this feature, so you may already be familiar with it.

To specify a region:

> Insert a #region keyword and a description at the beginning of your segment and a corresponding #endregion keyword at the end of your segment (see Figure 3).

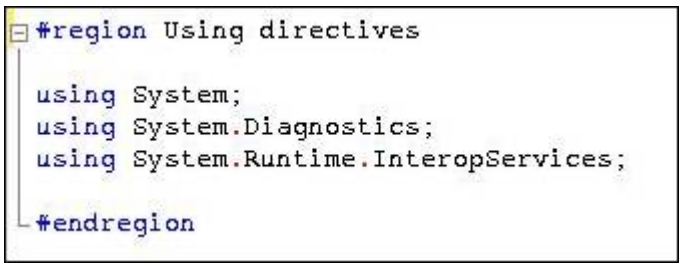


Figure 3 - Creating regions

The Outlining menu displays various collapse and expand options. To expand and collapse the current region you are in:

> Press Ctrl-M, Ctrl-M.

To expand or collapse all regions at once:

> Right-click the gray bar to the left of the main editor window.

To collapse an individual region:

> Click the plus sign next to the #region keyword.

This collapses the code into a single line that shows the region description.

To display the inside of a collapsed region:

> Move the mouse over the gray description area (see Figure 4).

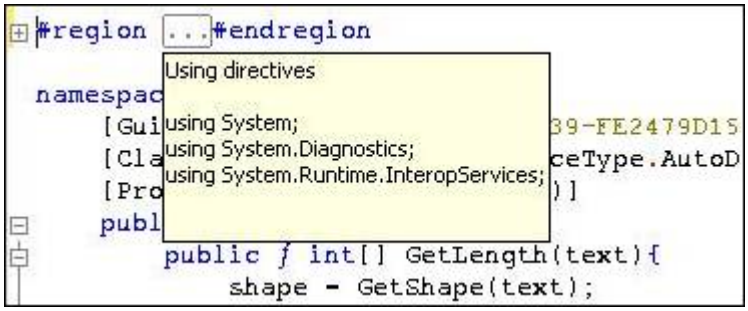


Figure 4 - Mouse over a region to see its content

You can even drag and drop collapsed regions inside your code. When you paste a collapsed region into a different location, the pasted text is automatically expanded.

1.3 -Hiding Selection by Using Temporary Regions (Ctrl-M,Ctrl-H)

Regions are created automatically for methods, comments, and sections encompassed by the #region compiler directive. In Visual APL and in regular text files, you have the option to create temporary regions around any section without the need for "#region". This is useful when you want to create a region that will not be preserved once your project is closed. In this case, you can temporarily define a region using the following method:

Highlight the section you want to hide and press Ctrl-M, Ctrl-H.

This hides the current selection in a temporary collapsed region (see Figure 5).

To expand a temporary collapsed region:

Press Ctrl-M, Ctrl-M.

Temporary regions are lost after you close a project.

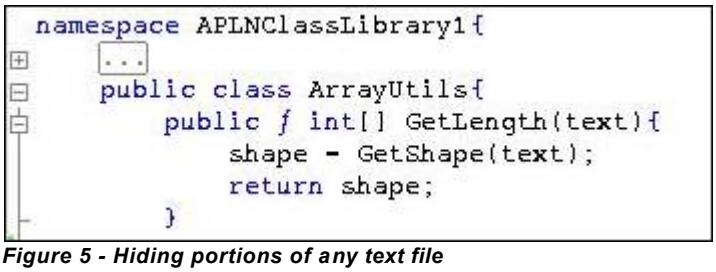


Figure 5 - Hiding portions of any text file

For creating regions which are preserved past the closing of your project, see "Creating Regions"

1.4 -Selecting a Single Word (Ctrl+W)

To select a single word when editing code:

Double-click anywhere in the word or just press Ctrl-W.

Double-clicking in a word is a common method used by many word processing and publishing programs to quickly select a word.

1.5 -Placing Code into the Toolbox (Ctrl-Alt-X)

When creating a project, you may want to use certain pieces of code or text again and again. You may have a standard copyright header that you place at the top of each file or a certain line of code to perform a common task. To simplify this repetitive task, you can place it into your Toolbox. The Toolbox is the window that lists all windows or web controls. To place your item into the Toolbox, use the following method:

1. Pull up the Toolbox:

   Press Ctrl-Alt-X.

2. Move the frequently used text or code into the Toolbox:

   Highlight your item and drag the selected text onto the General tab in your Toolbox (see Figure 10).

3. Rename the produced text item in your Toolbox

   Right-click it and choose Rename Item from the pop-up menu.

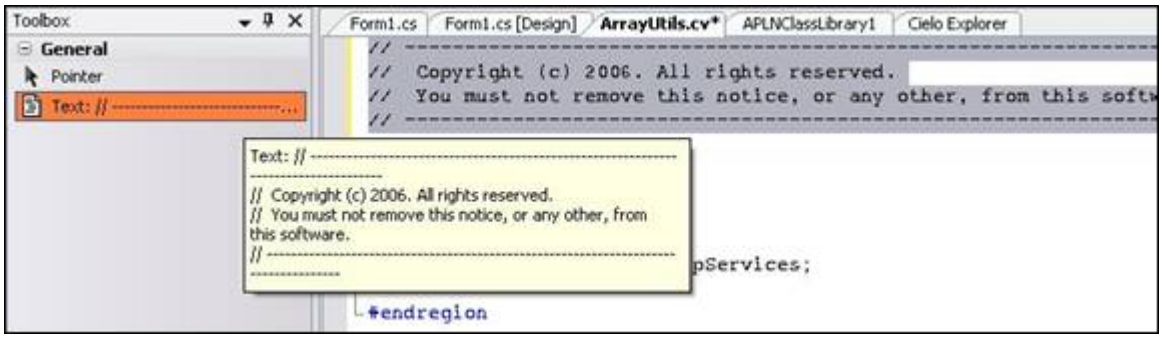


Figure 10. Adding text to the Toolbox

To insert an item into your text or code from the Toolbox:

Select the item in your Toolbox, then either:

Drag it into your code window or

Place your cursor in your text where you want to insert the item,
Double-click the entry

The General tab in the Toolbox is project and solution independent, and it retains its content even after you restart VS.NET.

1.6 - Using the Clipboard Ring (Ctrl-Shift-V)

The Clipboard Ring works like a historical file of the last used text selections that you placed on the Clipboard. Because it preserves many levels of selections, it is useful when you accidentally overwrite the current Clipboard content or when you find yourself needing several different items concurrently.

To use the Clipboard Ring you can either:

Double-click one of the remembered Clipboard items to paste it at the cursor's current location or

Drag it into the editor.

When the Clipboard Ring contains many Clipboard items, or when you cannot see the complete contents of each item in the ring, it's useful to cycle through the Clipboard Ring.

To progress one item at a time through the Clipboard Ring:

Select Edit > Cycle Clipboard Ring (Ctrl-Shift-V)

Doing this repeatedly makes VS.NET cycle through the Clipboard Ring's contents, displaying the stored Clipboard contents in the text editor at the cursor's current location. This method makes it easy to paste specific content in the code editor as it becomes visible during the cycling. Continue cycling through the Clipboard's contents until you find your desired item.

1.7 - Transposing a Single Character or Word (Ctrl-T or Ctrl-Shift-T)

To switch the position of the current characters or words on either side of the cursor you need to use Transpose. This procedure switches the characters or words, then moves the cursor to the right. Transpose is useful if you mistype a word or write a sentence or code segment with words in the incorrect order.

To transpose a single character:

Press Ctrl-T.

This swaps the two characters surrounding the cursor and moves the cursor to the right by one character. Pressing Ctrl-T repeatedly allows you to move a single character further to the right one character at a time.

To transpose a single word:

Press Ctrl-Shift-T.

Note: This does more than just swap two adjacent words. VS.NET knows to ignore "unimportant" single characters, such as equal signs, string quotes, white spaces, commas, etc.

Suppose you have a line of code that originally looks like this:

```
new SqlCommand( trans , stored_procedure, conn);
```

Pressing Ctrl-Shift-T repeatedly on the word "trans" would yield the following:

```
new SqlCommand( stored_procedure , trans, conn);        "
```

and finally this:

```
new SqlCommand( stored_procedure , conn, trans);        "
```

The quotation marks and commas retain their original positions throughout the process. When you reach the end of a line, pressing Ctrl-Shift-T transposes the word with the first word of the next line.

1.8 - Cutting, Copying, Deleting, and Transposing a Single Line

If you need to cut, copy, delete or transpose an entire line, you can do this easily with one keyboard sequence.

To **copy** the complete, current line to the Clipboard:

> Press Ctrl-C (or click the Copy icon) without any text selected.

To **cut** an entire line:

> Press Ctrl-X (or click the Cut icon) without any text selected.

This will cut the entire current line and place it in the Clipboard.

To **delete** a single line:

> Press Ctrl-L without any text selected.

To **transpose**, or **swap** the current line with the one below it:

> Press Alt-Shift-T without any text selected.

Doing this also moves the cursor down by one line. This allows you to press this keyboard shortcut repeatedly until you move your current line to the desired position.

1.9 - Formatting Entire Blocks (Ctrl-K, Ctrl-F or Ctrl-K,Ctrl-D)

To apply formatting to an entire selection, there are several useful functions you can use. Uppercasing, lowercasing, or deleting horizontal white spaces are just a few examples.

To access these features:

Select Edit > Advanced

One of the most useful features here is the Format Selection function.

To access the Format Selection Function:

Press Ctrl-K, Ctrl-F,

This feature formats an entire selection and inserts tabs where appropriate to modify the code with the correct code-specific block indentation. This is usually done automatically when someone enters code upon closing a block (such as by typing the "}" sign in Visual APL) but Format Selection forces this automatic format (see Figures 11 and 12).

You can also format the entire document. To do this:

Press Ctrl-K, Ctrl-D.

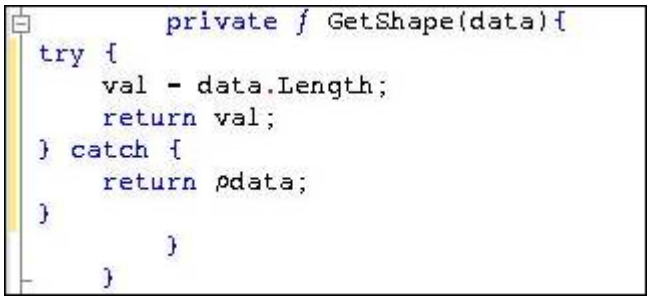


Figure 11. Before formatting block

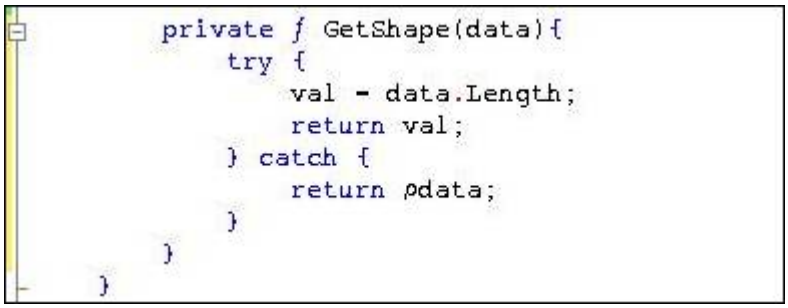


Figure 12. After formatting block

1.10 - Toggling Word-Wrapping (Ctrl-R,Ctrl-R)

To turn word wrapping on and off for the current view:

Select Edit > Advanced

Or use the keyboard shortcut:

(Ctrl-R, Ctrl-R)

1.11 - Creating GUIDs

As you develop new classes and components, you often need to create Global Unique Identifiers (GUIDs). These are 128-bit values often represented by 32 hexadecimals. In the past, component developers used GUIDs to assign their components with unique names to reduce the likelihood of two components sharing the same GUID. Developers now use GUIDs for anything that requires a unique identifier. GUIDs can be created manually by randomly selecting 32 hexadecimals, but this is somewhat tedious. VS.NET comes with a utility that creates GUIDs for you whenever you need one.

To create a GUID, open the Create GUID dialog box:

Select Tools > Create GUID (see Figure 13).

Here you can generate identifiers in various formats, including common code items often used in COM development.

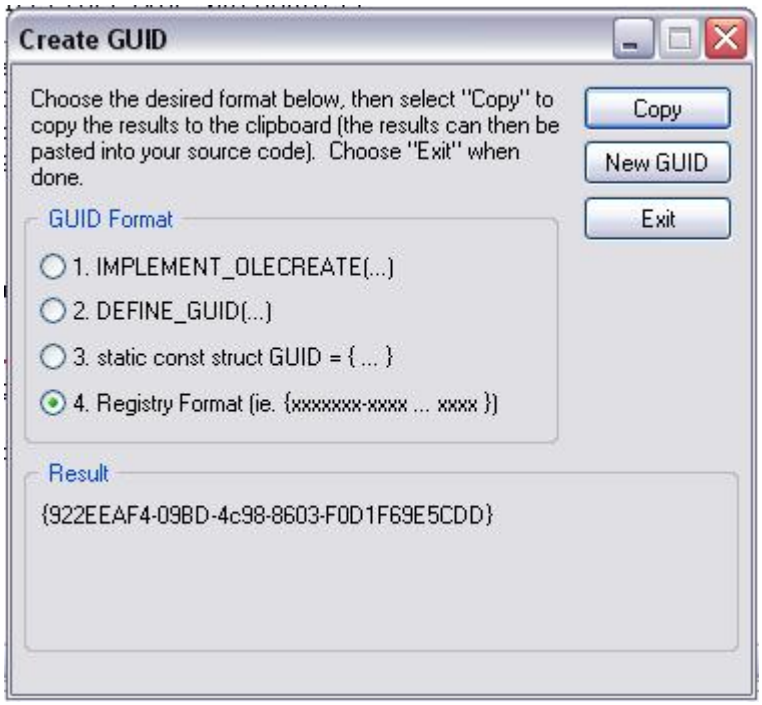


Figure 13. Create GUID tool

1.12 - Creating Rectangular Selections

To make a rectangular selection of text or code, there is a technique which allows you to do this without including the intervening lines (see Figure 14).

To select a rectangular area:

Press the Alt key while dragging the mouse to select the area.

Manipulating the selection by copying, cutting, or pasting rectangular blocks can be done very easily this way.

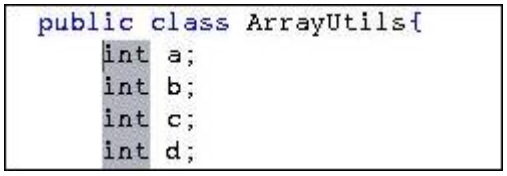


Figure 14. Rectangular selection

1.13 - Switching Between Views (F7)

For Windows forms, you can easily switch between both views. To toggle between designer and code views:

Press F7 (designer and code)

1.14 - Going to a Line Number (Ctrl-G)

For quick and easy navigation inside your code or text file, you can jump to a particular line.

To go to a specific line number you need to access the go to dialog box. To do this, either:

press Ctrl-G or

double-click the line number status bar at the bottom.

A small dialog box will appears. To jump to a line number:

enter a line number

If you type a line number that is out of the range of possible line numbers, the cursor jumps to the beginning or end of the file, respectively. A number which exceeds the number of lines in the file will place you at the end of the file. A number that is too low will jump you to the beginning of the file.

1.15 - Searching for a Word

There are several methods for searching for a word inside a file. It is helpful to know all the methods for ease in moving around your file. The following are common methods for finding a word.

1. Access the Find dialog box

   Select Edit > Find

   Enter a term in the Find dialog box.

2. Access the Combo box in the main toolbar next to the configuration drop-down list. To open the combo box and invoke the search function:

   Press Ctrl-D

   Enter or paste a word into this list and press Enter

   Repeat pressing Enter in that drop-down list to find the next match.

3. Select the entire word, or place the cursor somewhere inside the word:

   Press Ctrl-F3.

   This invokes the same search function I described just previously. Repeatedly pressing Ctrl-F3 iterates over all matches.

Using either the combo box or the Ctrl-F3 shortcut applies the same search options specified in the Find dialog box. Set the options you desire in the Find dialog box first to search correctly (for example, enabling Search Hidden Text to include all collapsed regions in the search area).

1.16 - Performing an Incremental Search (Ctrl-I)

An incremental search allows you to find occurrences of a search key as you type it one letter at a time. After each keystroke, VS.NET immediately highlights the next available occurrence that matches whatever you have typed so far. The more letters you type, the more likely is it that the found occurrence is indeed what you are seeking.

To initiate an incremental search:

Press Ctrl-I

You do not need to enter the entire word to find a specific occurrence; you only need to type the minimum number of characters that would uniquely identify the word for which you are searching.

To return to normal editing mode:

Press Escape

In the Cielo Explorer you have to single click with the mouse.

If you are unsatisfied, press Ctrl-I repeatedly to find the next occurrence that matches your partial search key, or press Ctrl-Shift-I to find the previous matches. You can, of course, simply enter more letters to narrow the search further.

1.17 - Searching or Replacing with Regular Expressions or Wildcards

Regular expressions can look extremely intimidating, but they are extremely powerful tools to find complicated search keys and patterns. Regular expressions is a built in feature that allows you to describe a searchable pattern in terms of wildcards, characters, and groups.

This feature in VS.NET is often overlooked by many developers. To access this feature, bring up the Search or the Replace dialog box:

> Press either Ctrl-F or Ctrl-H, respectively

Note: Besides the regular options to refine your search, the last check box allows you to define your search based on regular expressions or wildcards.

You can use either of two modes. To use Regular Expression mode, specify the expression using a similar notation you are accustomed to with the System.Text.RegularExpressions namespace.

To see a list of possible constructs that you can insert into your regular expression:

> Click the arrow button next to the Find What field (see Figure 15).

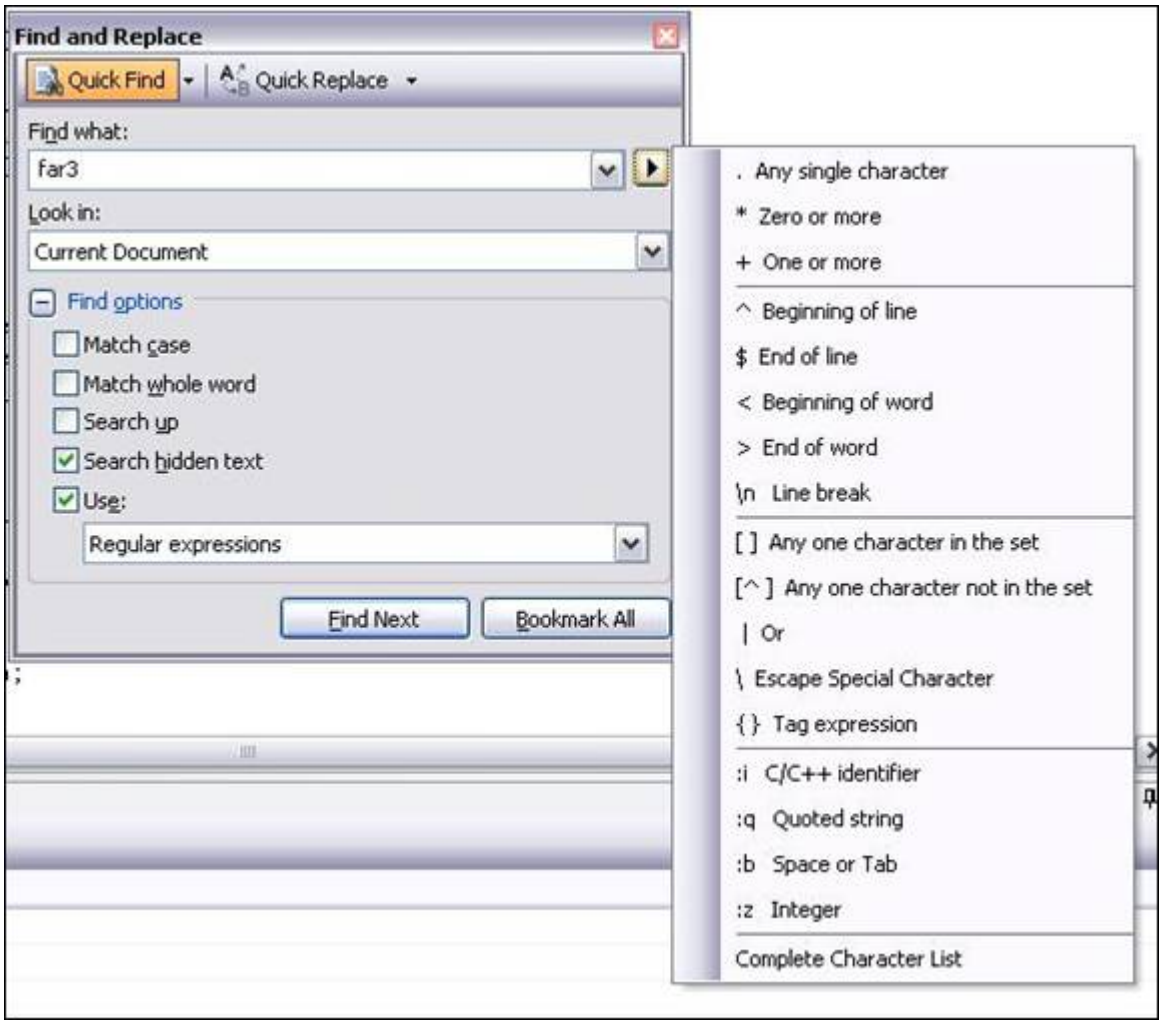


Figure 15. Use regular expressions.

To use Wildcard mode, construct your search pattern using the more commonly known wildcards from MS-DOS, such as "*" and "?".

If used correctly, these two modes can be very helpful in refining your search algorithm or when developing programs based on regular expressions.

1.18 - Global Search or Replace (Ctrl-Shift-F or Ctrl-Shift-H)

The global search and replace feature in VS.NET spans entire projects and solutions. This is similar to normal search and replace dialog boxes, except that you can specify the scope of the search or replace action over multiple files.

To bring up the global search or global replace dialog box:

Press Ctrl-Shift-F or Ctrl-Shift-H, respectively

This feature allows you to perform a global search and replace in just the current document, the current project, the entire solution, or any open documents (see Figure 16). You can also filter which files you want to search based on wildcards.

Figure 16. Global search and replace.

Once the search or replace action is started, VS.NET searches all specified documents and modifies them if required. The global replace, will also prompt you to leave modified documents open. This option allows you to undo the replace, because only open documents offer the undo feature. If you don't select that option, global replace will automatically save the modified files and make this a permanent action.

To immediately stop a global search or replace anytime:

Press Ctrl-Break.

Once a search or replace action is completed, a list of occurrences that have been found will be displayed in the "Find Result" dialog box.

To iterate over the Find Result list:

Press F8 or navigate to an occurrence by double-clicking it.

If that occurrence is currently located in a collapsed region, you can expand it. To automatically expand the region:

Double-click the same find result in the list again.

On initiating a new find or replace action, VS.NET clears this window to fill the list with the new results. If you want to keep the results of the previous search and output the result in a second window:

Check the Display in Find 2 option in the search dialog box

You can then tab between both result sets.

All find/replace functionalities are included in a single dialog box (see Figure 17), you can also access the global find/replace functionalities using the drop-down list at the top. All shortcuts remain the same.

Figure 17. Global replace in files

1.19 - Using Bookmarks

Bookmarks enable you to return quickly to a given page or section of your code or file. When you determine critical sections of your programming that you want to return to frequently, instead of scrolling to these places, bookmark those lines.

To place a bookmark, first, make the bookmark toolbar visible:

Right-click any existing toolbar and select Text Editor from the pop-up menu.

Click on the blue flag icon in Text Editor toolbar.

Another method which can be used to place a bookmark:

Press Ctrl-K, Ctrl-K.

This second method not only makes a bookmark visible on the left side of the code, but you can now jump quickly among other bookmarks. To jump to the other bookmarks you can either:

Click the appropriate flag buttons on the toolbar (see Figure 18) or

Press Ctrl-K, Ctrl-P (for the previous bookmark) or Ctrl-K, Ctrl-N (for the next bookmark).

Figure 18. Bookmark toolbar

To clear all bookmarks:

Press the Clear Flag icon or

Press Ctrl-K, Ctrl-L.

The Find dialog box in VS.NET allows you to bookmark all matching occurrences as follows:

Click the Mark All button.

As part of VS.NET 2005 considerable support for bookmarks, you can also have the option of moving to the next or previous bookmark within the same file as follows:

Press the appropriate buttons on the bookmark toolbar (see Figure 28).

Figure 28 - Move to bookmarks in the same file in VS.NET 2005

You can also name your bookmarks by first opening a new Bookmarks window:

Press Ctrl-K, Ctrl-W or

Select View > Other Windows >Bookmark Window.

This displays all the bookmarks that you have created (see Figure 19).

To jump to a bookmark's location:

Double-click the bookmark.

To rename a bookmark:

Press F2 or

Right-click the bookmark and use the Rename context menu item.

You can categorize your bookmarks and organize them into folders. You can also, jump to the next or previous bookmark within the same folder. To perform any of these functions, simply select the appropriate icon on the toolbar.

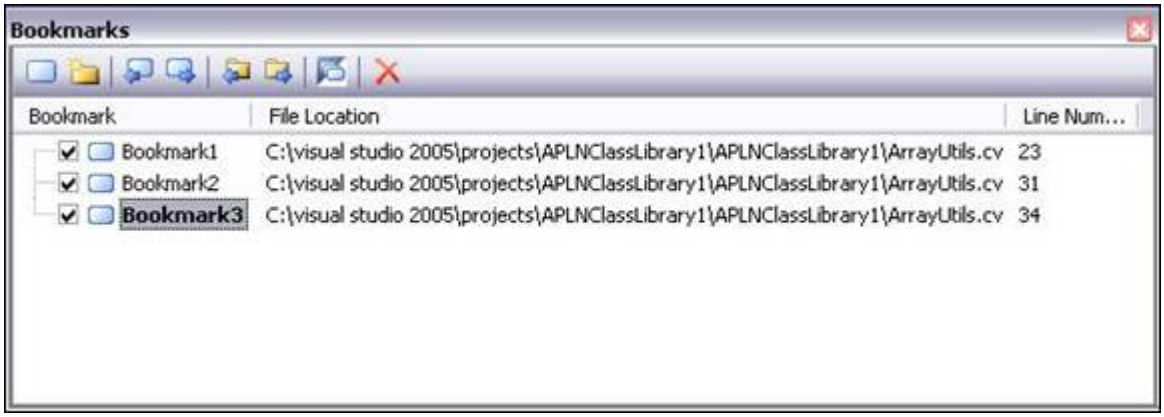


Figure 19. Manage your bookmarks in the Bookmarks Window.

The Bookmarks window shows check boxes next to each folder and bookmark. These allow you to disable a bookmark without deleting it. Disabled bookmarks are skipped when you use any of the buttons or shortcuts to navigate your bookmarks.

1.20 - Using Browser-Like Navigation (Ctrl -, Ctrl Shift -)

VS.NET is equipped with browser-like "back" and "forward" buttons in the IDE that allow you to review the most recent cursor locations. The Navigate-Backward and Navigate-Forward buttons are located to the right of the Undo and Redo buttons (see top left of Figure 20). You can also access them in the View menu.

Figure 20. Navigate buttons

Similar to a web browser, VS.NET keeps a history of your recently accessed locations. After using the Go To Definition feature or after switching arbitrarily to another file or even just jumping between different line numbers of the same file, you can easily return back to the last edit location as follows:

Click the Navigate-Backward button.

These Navigate buttons have pre-assigned shortcuts. To Navigate back:

Press Ctrl-Hyphen

To Navigate forward:

Press Ctrl-Shift-Hyphen.

1.21 - Inserting External Text File

A common method of inserting code fragments involves opening a file in Notepad and copying the code from there. To bypass this step of opening and closing Notepad you can:

Select Edit > Insert File as Text from within the code editor.

Chapter 2: Exploring the IDE

Visual Studio .NET is an easily customizable feature-rich Integrated Development Environment (IDE). It allows a developer quick access to commonly used commands and activities which enable you to control and modify your project and solutions. This chapter covers a range of topics such as: the Solution Explorer; window positioning; managing macros; modifying menu items and other tips and tricks useful for in navigating inside the IDE.

2.0 - Setting Project Dependencies

In a large solution with multiple projects and custom build events, it is often necessary to control the build order for your projects. VS.NET has the capability to figure out which project needs to be built first by analyzing the references of each one. The first project built is normally the one referenced first. This algorithm is based on your set project references for your projects.

VS.NET also allows you to compile a certain project before another one without having project references. This is accomplished from a pop-up menu that allows you to choose Project Dependencies. To designate the order in which projects will be built:

Right-click your project that needs to be built last and choose Project Dependencies from the pop-up menu.

Set manual dependencies on other projects by check-marking them.

This will ensure that the checked projects will be built before the current project (see Figure 21). A drop-down menu for the current project allows you to switch to another project's dependencies.

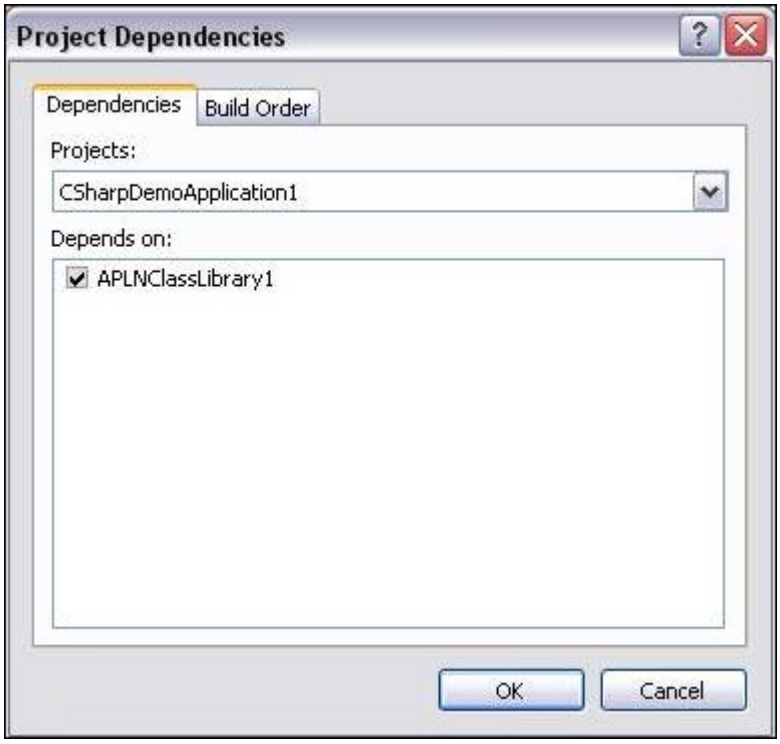


Figure 21. Setting project dependencies manually

VS.NET prevents you from creating circular references or modifying dependencies that resulted from adding project references. To verify the build order at any given time:

Click the read-only Build Order tab.

Note: In VS.NET 2005, the Project Dependencies context menu item in the Solution Explorer does not exist for web applications, instead, you need to select Websites > Project Dependencies.

2.1 - Embedding Files As Resources

Embedding files as resources allows you to embed any given file directly into your produced assembly. For instance to display a company logo on your Windows application, you could produce a regular Windows assembly and link to an external image that you send along with your application. You can also embed the image right into the assembly you produce. This enables you to avoid shipping the external image and, more importantly, prevents the possibility of these two files becoming separated.

To embed a file as a resource, it must first be included in your solution. You can then select the file in the Solution Explorer and change the Build Action property in the Properties window. The build action tells the compiler what to do with the specified file. If you select the Embedded Resource build action, the actual bytes of the file will be stored inside the produced assembly (regardless of whether this is an EXE or a DLL).

At runtime, you can then extract the bytes using the following code:

```
Assembly oAssembly =
System.Reflection.Assembly.GetExecutingAssembly();

Stream streamOfBytes =
oAssembly.GetManifestResourceStream( mylogo.bmp );          "          "
```

After this retrieves the bytes from the given embedded resource, you have to convert those bytes back into the original file type (using Image.FromStream(), for instance, to convert it back into a picture). Notice how this code is orthogonal to the file type being embedded as a resource. This enables you to embed any file type: sound and movie files, PDF files, or even another assembly.

2.2 - Changing the Font Size of IDE Windows for Demos

It is a common practice when demonstrating VS.NET or your code, to increase the font size of the text editor so that everyone in the audience can easily see the demonstration. The font size can be easily increased by a couple of methods:

Select Tools > Options > Environment > Font and Colors > Size.

This works great, except that the text in the Output windows, Solution Explorer, Class view, Macro Explorer, or in the file tab titles can still be very hard to read.

Control the size of the text in these elements as follows:

In the same settings window, the first drop-down list reads Show Settings For. Change it to read Dialogs and Tools Windows.

Set the font and the size here in this window.

You control the format of the text elements of the majority of the IDE windows. The changes come into full effect after you restart the IDE.

To increase the font of the Output window:

Change Show Settings For to Text Output Tools Windows.

To reset any of these settings to their default installation values:

Click the Use Defaults button.

Note: This button applies only to the currently selected item in the Show Settings For drop-down list, so repeat this step for every setting that you want reset back to the default settings.

2.3 - Dragging Files to Obtain a Full Path

A useful feature of VS.NET is the ability to drag files from your Solution Explorer directly into your code. If you do this in a source code file, it will simply insert the full path to the selected file into your code.

2.4 - Moving Any Window Around

Every window in VS.NET is movable, resizable, and dockable: the Solution Explorer or Macro Explorer; the Properties, Task, and Output windows; and even your Toolbox, Server Explorer, and Find/Replace windows. To move any window in VS.NET:

Drag the title bar to the desired position.

As you drag a window close to a dockable region (such as tabs or near another window frame), an outline appears, allowing you to preview the result before dropping the window.

To dock and undock windows:

Double-click the title bar.

You can also move the order of tabs in your tab windows. This includes the files tabs at the top of your editor.

While the ability to control window positioning gives VS.NET enormous flexibility, the preview outlines are too confusing to make this an intuitive interface. If you have moved the windows positions and would like them reset, you can always reset all windows positions to their installation defaults:

Select Tools > Options > Environment > General > Reset Windows Layout.

With VS.NET 2005, you can also reset the windows positions. To do this:

Select Window > Reset Windows Layout.

One aspect of moving windows around is the ability to create a split screen. Use the following steps to split the editor into two vertical screens complete with their own set of file tabs (see Figure 23):

Drag the tab of any open file and move it to the right of your editor (to the left of where the Solution Explorer usually resides).

This docks your selected file to the right and splits the editor into two vertical screens.

To close vertical split mode either:

Close the second set by clicking the small X at the top right, or

Drag the file tabs back to the left along with the other files.

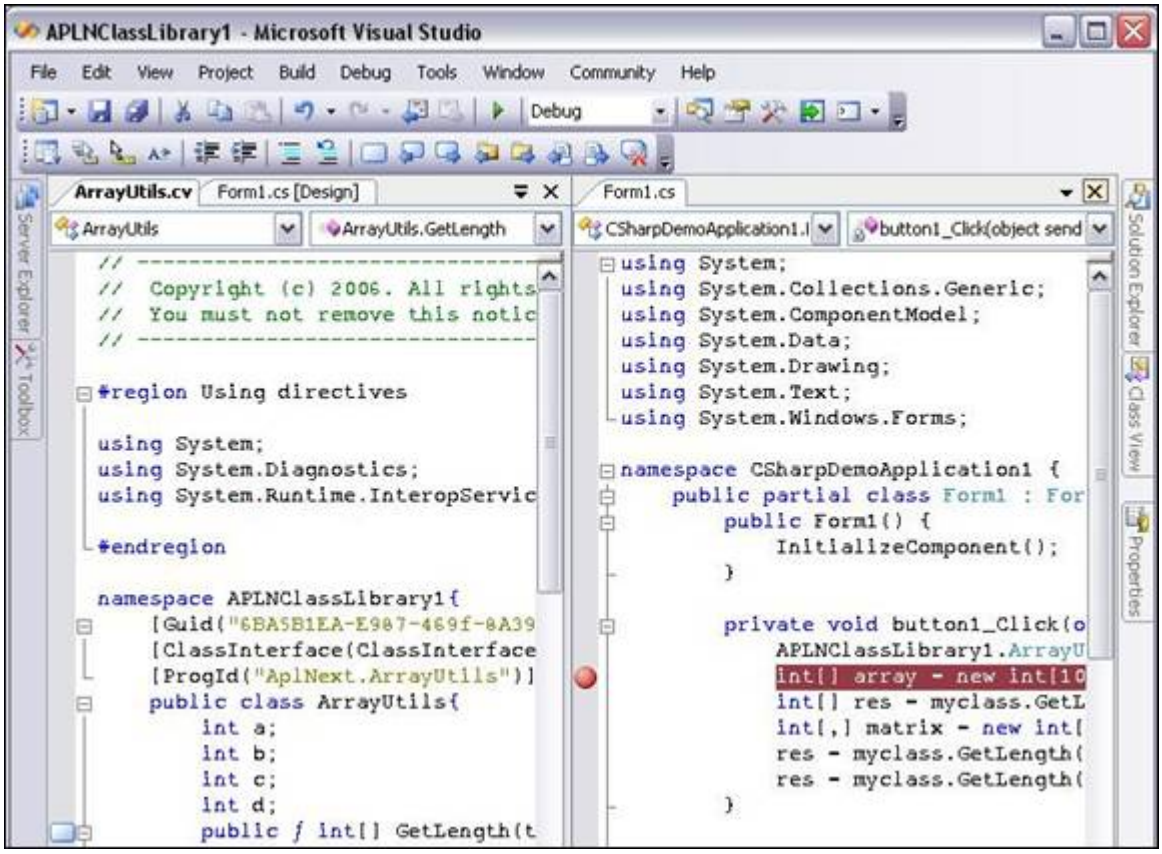


Figure 22. Vertical split

To create a horizontal split screen:

Drag a file tab to the bottom of your editor.

2.5 - Creating Split Screens in the Same File

The "Moving Any Window Around" trick described in "Moving Any Window Around" shows how to create split screens so you can see two files next to each other. What if you want to create a split screen to see two locations of the same file? To do this:

select Window > Split

The horizontal divider can also be generated using a faster method:

Move your cursor right above the vertical scrollbar of the main editor. There is a very thin, short, rectangular-shaped divider (see Figure 23).

Place your mouse over that divider, the mouse icon changes to the divider icon.

Drag the divider down to the center of the screen to create the split screen (see Figure 24).

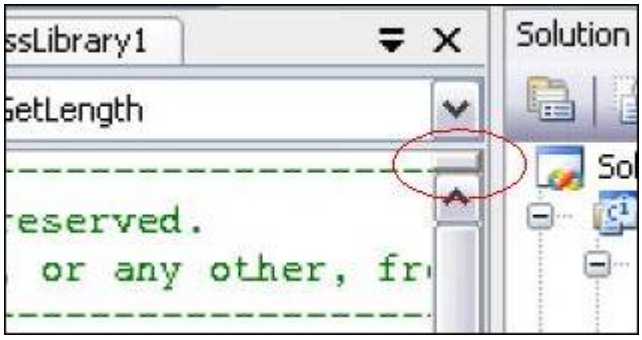


Figure 23. Horizontal split divider

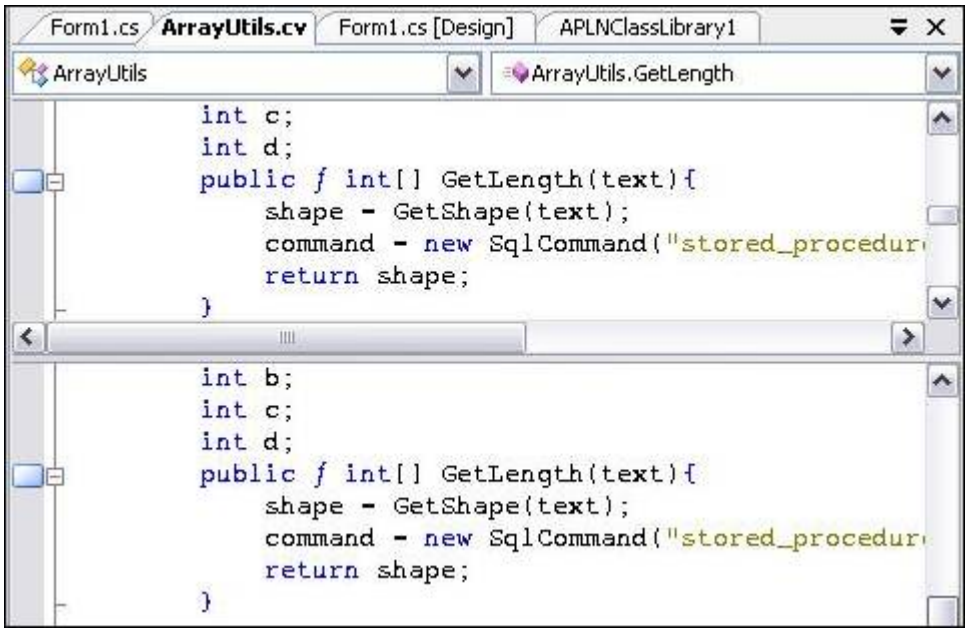


Figure 24. Split document.

To move the divider back to the top of your editor window:

Select the divider bar and slide back into its original position.

2.6 - Customizing the VS.NET Menu and Toolbars

The VS.NET menu can be customized in a variety of ways. You can add and remove commands as well as reorder them. To customize the menu and toolbars:

Select Tools > Customize.

With the Customize dialog box open, navigate back to the VS.NET menu.

The menu now does not react to left mouse-click events and will show context menus when you right-click the menu items. Here you can rename, edit, and delete menu items; drag menu items around; or even create your own cascading menu groups.

You can also manage the icons for each menu item by right-clicking the item and selecting Choose Button Image from the pop-up menu. If you are not satisfied with the icons in the selection, you can copy icons from other menu items to your newly created menu ones. To copy icons from other menu items:

Right-click a menu item with the desired icon.

Choose Copy Button Image from the pop-up menu

Right-click the menu item you want to modify.

Choose Paste Button Image from the pop-up menu.

To add other commands to a menu:

Drag a command from the Command tab directly into the VS.NET menu.

Note: In addition to directly modifying the VS.NET menu items as long as the Customize dialog box is open, VS.NET 2005 adds a complete new GUI to modify the menu. The new GUI appears when you select Tools > Customize > Rearrange Commands. Here you can move, add, and delete menu items as well as toolbar buttons (see Figure 25).

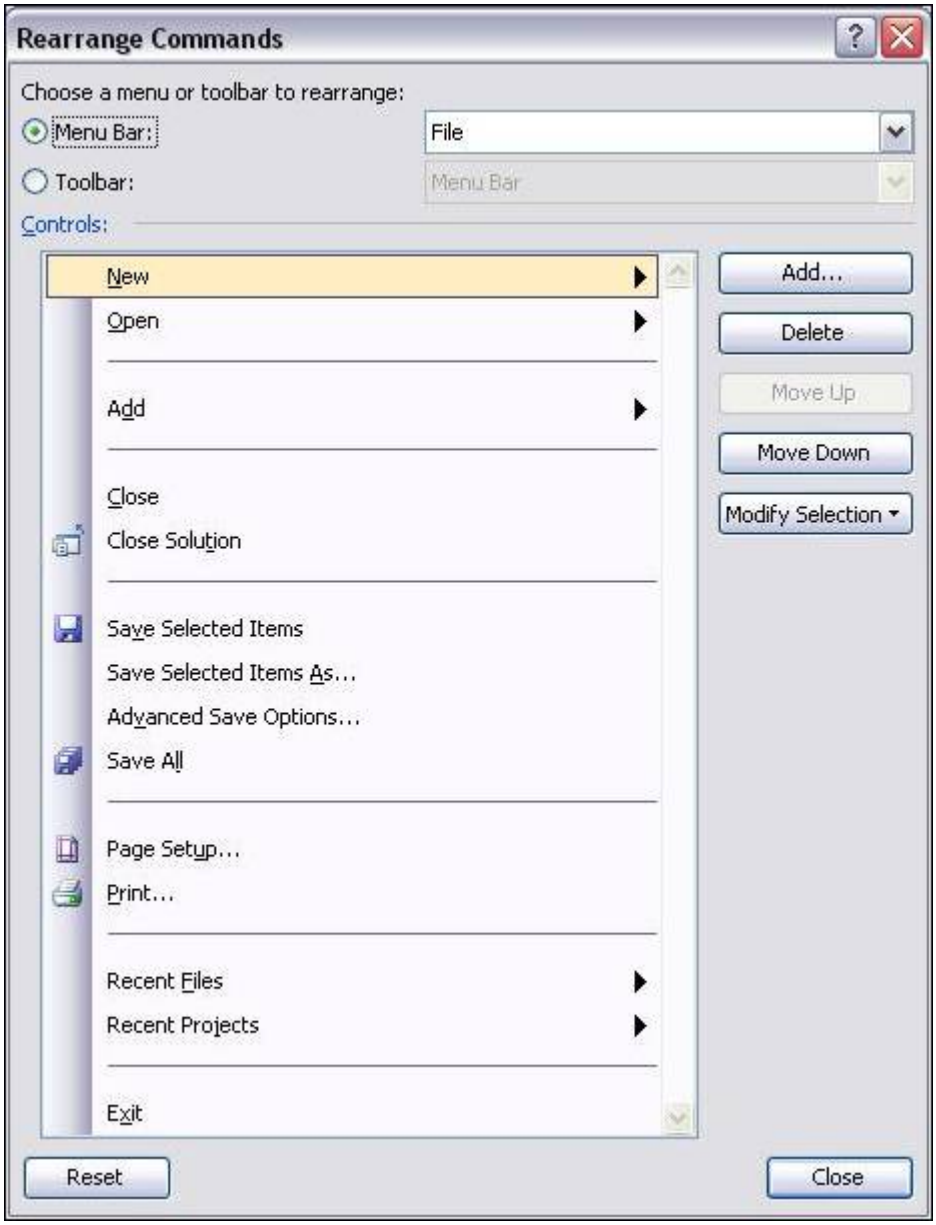


Figure 25. Customize menus using the rearrange commands

2.7 - Dragging Files from Windows Explorer into VS.NET

Visual Studio .NET completely supports file drag (and drop) actions. It allows you to drag files from Windows Explorer directly into VS.NET. If you drop them in the Solution Explorer under a project, it will first be copied into the same directory as the project and then included into the project. If you drag them into the code editor, VS.NET will either start the default external viewer (for example, Adobe Acrobat for PDF files) or display the file's contents inside VS.NET if it's a file type that it understands.

To drag files from Windows Explorer into VS.NET if you don't have enough screen space:

Drag the file into the Windows taskbar at the bottom of your screen

Pause for a few seconds over the taskbar for VS.NET. The pause brings VS.NET into focus.

Drop the file into the appropriate location.

2.8 - Using Full-Screen Mode (Ctrl – Shift – Enter)

Full-screen mode allows you to hide virtually everything except the main editor, where the entire screen shows the main view. To enter full-screen mode:

Select View > Full Screen or

Press Ctrl-Shift-Enter.

The main menu is still visible at the top, and a floating button that closes full-screen mode is also available. To hide the Close Full Screen mode button—you need to memorize the keyboard shortcut that returns to normal mode or:

Select View > Full Screen again.

Full-screen mode is available for any view, including the HTML, Designer, and XML views.

2.9 - Copying the Fully Qualified Name of a Class

The Class view is a hierarchical view of all your classes and namespaces in your solution. To display this view:

Select View > Class View or press Ctrl-Shift-C.

To go to any class and its members and navigate to the member definitions:

Double-click on the desired item.

Another useful feature allows you to extract the full namespace of any class or member:

Highlight the class or the class member.

Press Ctrl-C.

This copies the complete namespace of the selected item to the Clipboard. This feature comes in handy when you have a complex or deep namespace structure.

To paste the namespace into the VS.NET code editor, there is no need to copy it to the Clipboard first. Use the following method:

Drag a class or member of a class from the Class view directly into your code

Watch VS.NET paste the complete namespace and member name there.

2.10 - Changing Properties of Several Controls

When designing your Windows forms, you can use the Properties window to modify a control's behavior and appearance. The Properties window, however, is adaptable when you select several controls at the same time. To select a series of controls either:

Hold down Ctrl or Shift when selecting controls or

Draw a selection rectangle with your mouse,

The Properties window automatically displays the properties that are common to all of the selected controls. With all controls selected, any change you make in the Properties window affects all selected controls.

This is useful for instance, after you drag a series of text boxes from the Toolbox onto your form and want to get rid of the default "TextBox1," "TextBox2," etc. values.

Select all the text boxes.

Change the Text value to a single space by pressing Spacebar.

Change it back to an empty string by pressing Delete.

Do this twice because the initial values of each text box differ originally, so the Text property displays an empty string as the "common value".

This deletes the default text in all of them.

2.11 - Locking Controls

When laying out windows controls on Windows forms, you can easily move the controls around or create event handlers by simple dragging and double-clicking. However, this simplicity has its drawbacks as you can move things around accidentally very easily. This can cause problems if you have already finished designing your Windows forms. In order to prevent this from happening, you can lock your form.

To lock the position of your controls on your form:

While in the Designer view, right-click anywhere on your form

Choose Lock Controls from the pop-up menu (see Figure 26).

You still have the ability to add event handlers and modify a control's appearance, but you can no longer accidentally move or resize a control. To indicate that it is locked and unmovable, a thin, black outline appears around each selected control.

To return to the Designer view:

Right-click your form and choose Lock Controls from the pop-up menu again.

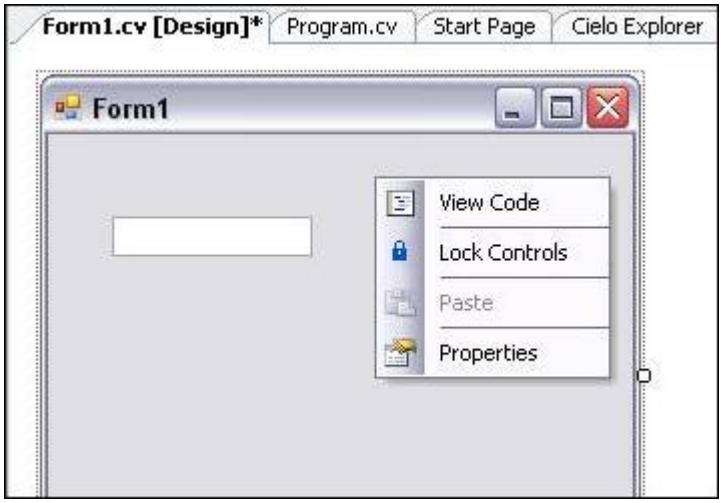


Figure 26. Lock controls in a form

2.12 - Toggling the Description in the Properties Window

The Properties window not only displays all properties of a selected control, but the Description pane at the bottom briefly describes the active property. As you select different properties, the Description box informs you what the selected property does. To turn off the Description box panel:

Right-click the property name.

Choose Description from the pop-up menu.

To turn it back on use the same method.

2.13 - Change Drop-Down List Values in the Properties Window

Whenever a property only accepts a finite set of values, the value field becomes a drop-down list, from which you make your selection. For instance, the FormBorderStyle property of a Windows form only accepts None, FixedSingle, Fixed3D, FixedDialog, Sizable, FixedToolWindow, and SizableToolWindow. To select the appropriate item:

Open the drop-down list.

Select the style you want.

Anytime you have a drop-down list in the Properties window, you can iterate over the list more quickly by simply double-clicking the property or its corresponding drop-down list. Without expanding the list first, double-clicking it sets the value to the next available item in the list (or to the first item if the current value is the last one).

This trick can be extremely useful when switching Boolean values because a double-click changes the value quickly from True to False, or vice versa.

2.14 - Adding and Removing Event Handlers Through the IDE

Adding default handlers through the IDE is quite easy. In most cases, you only need to double-click a control which creates the necessary code for the default event handler.

Adding and removing non-default events handlers is still easy, but, in Visual APL, it requires not only the removal of the method itself but the removal of the code that hooks an event handler to an event, often found in the InitializeComponents() method.

The proper, but relatively hidden, way to add and remove event handlers in Visual APL is to use the Properties window:

- Select the control
- Click the Events button in the Properties window (the yellow thunderbolt).

The Property window displays all the events that the selected control exposes, along with any event handler that is already hooked up to them.

In addition, the event handler fields are clickable (see Figure 27).

To create an event handler:

- Double-click an empty field.
- Choose which event you want to subscribe to.

To hook an event handler which is already written, to an event:

- Use the drop-down button next to the selected field that automatically lists all matching event handlers.

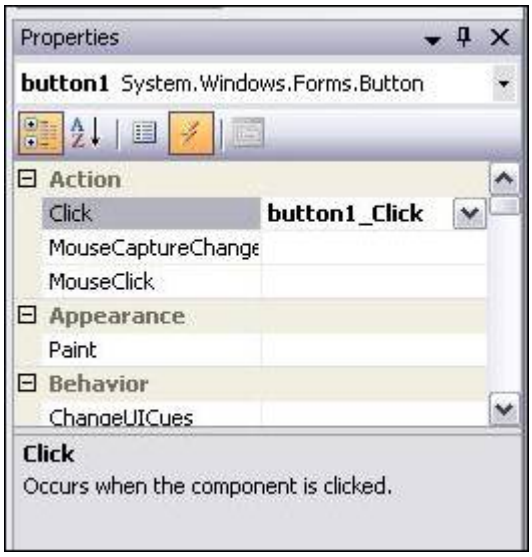


Figure 27. Setting events

To delete an event handler:

- Delete the value in the event field.

This also removes the event handler subscription you have in the InitializeComponents() method.

2.15 - Selecting Control Through a Drop-Down List

When there are many controls on a Windows form, it can become a challenge to find a specific control, and select it. This problem often occurs when many panels overlap one other or when the Windows form becomes too crowded to isolate a specific control that you want to modify.

To select a specific control:

Select the drop-down list that appears right above the Properties window.

Select the desired control.

Note: This drop-down list is only populated in the Designer view. It contains all the controls that exist on the Windows form. To select a certain control, you just need to know its ID and data type.

Chapter 3: Compiling, Debugging, and Deploying

Not only is VS.NET a great editor, it is also a powerful compiler, debugger, and profiler. It allows you to precisely control your compilation procedure and provides the features which are absolutely essential in to locating and fixing a bug: analyzing your code, attaching to running processes that you want to debug, and changing code and variables at runtime. This chapter covers topics that you need to know when it comes to compiling and debugging your programs.

3.0 - Setting the Default Namespace and Assembly Name

Following the official naming guidelines suggested throughout the industry, you would declare your classes in your own company and project-specific namespace. Typically, you end up with the following namespace hierarchy (at a minimum):

```
MyCompanyName.MyProject.MyClass
```

When you add new classes with the Add New Item dialog box, VS.NET does not place your new class in any project namespace. It places it, by default, in the top-level namespace, which usually means the name of your assembly. To set the default namespace when you create new projects:

Select Project > Properties > Application.

Specify the default namespace in the Default Namespace field.

This namespace can be many levels deep; new classes added through the VS.NET dialog box will be placed in that specified namespace. In addition, you can also control the name of the assembly that is being generated by specifying it in the Assembly Name field. While Windows applications typically use one word for the assembly name, Control Library projects should be named using the same guidelines as the namespace.

3.1 - Generating Compiler Warnings Through the Obsolete Attribute

A commonly used way to display warnings in VS.NET at compile-time is to set an Obsolete Attribute to a method. Throughout the product development cycle, occasionally certain methods become obsolete. Sometimes the old method is not useful anymore. It may have become inefficient, or has been replaced by another method. If you can't modify those methods, will need to write another implementation of the method using a slightly different name or signature. To maintain compatibility, you do not want to remove the old method and break your code. This is where the Obsolete attribute comes in handy:

```
[Obsolete( Use the new MyMethodEx"instead  )]            !       "
public void MyMethod()...
```

Setting the Obsolete attribute as above makes a warning message appear in the Task List stating that the particular call to a method is obsolete. The warning message also includes your personalized message that you pass as the attribute's argument (such as, "Use the new MyMethodEx instead!").

As with the warning compiler directives, this method does not affect the compilation behavior in any way. You also must activate the Task List to see these warnings. Unlike warning compiler directives, the warning only appears if there is code that tries to invoke the obsolete method. These warnings will never appear if you don't refer to these methods anywhere in your code.

3.2 - Setting the Assembly Output Path

When you build a project, the produced assemblies are typically placed in the \bin\Configuration subfolder of your project folder, where the configuration folder is typically Debug or Release.
These are the default settings. To specify another directory where you want to place the produced assemblies and external files:

Select Project > Properties > Build for Visual APL projects.

Place either a relative or absolute path in the Output Path field.

This setting is used at the next build.

These configuration-specific properties allow you to specify a different output path for each configuration. For instance, if you want, you can set the default output path for the Debug release as the usual bin subfolder, while directing the release build directly to a network share on your internal company network.

3.3 - Setting the .NET Framework Version for Your Assembly

A great side-by-side installation feature of the .NET Framework is the ability to have multiple versions of the .NET Framework installed on a given computer, without any of them interfering. By default, all non-web applications use the .NET Framework with which they were compiled (if available), whereas web applications by default always use the most recent version of the .NET Framework.

You can specify which .NET Framework is supported and required for your assembly by modifying the application configuration file (MyApplication.exe.config or Web.config). What you need to do is:

Insert the appropriate Configuration/startup/supportedRuntime and Configuration/startup/requiredRuntime XML tags in the configuration file

Set its version attribute to the specific .NET Framework version.

This enables you to force a Windows application to use an older version of the .NET Framework.

This configuration modification is easy in VS.NET. To set this for Visual APL:

Select Project > Properties > General > Target Platform.

Set the supported and required runtime versions for your assembly (see Figure 28).

To verify that your assembly is picking up the correct version, check the Version property in the .NET Framework class System.Environment.

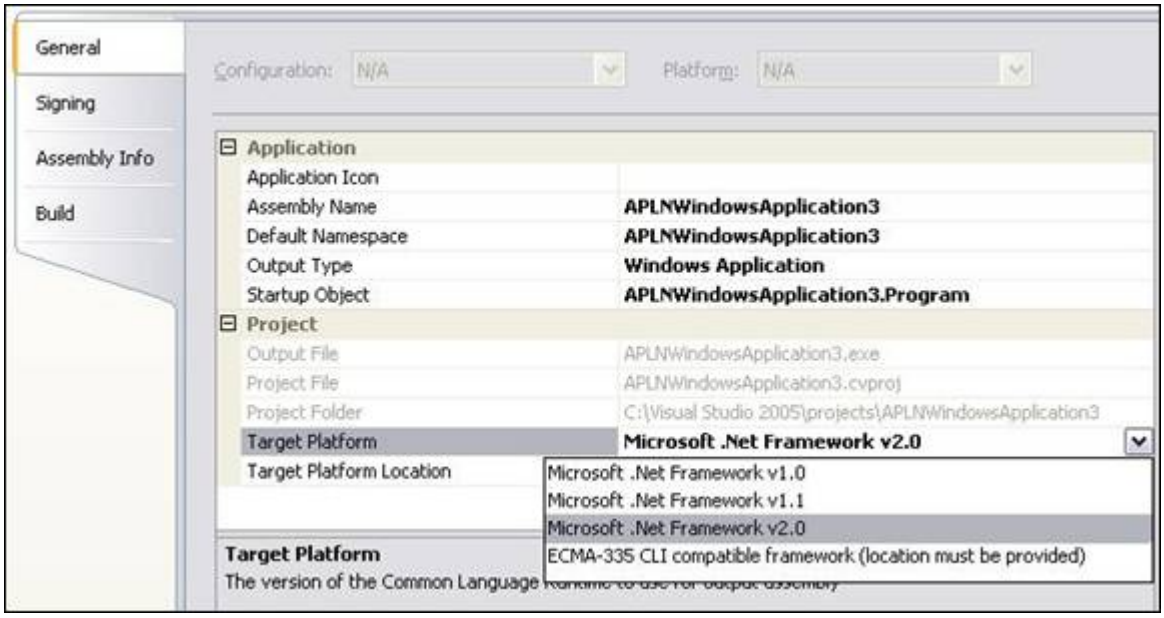


Figure 28. Choosing the target runtime.

Note: Supporting the 1.0 Framework, or even version 1.1 is an unsupported environment. Simple programs most likely will work, but for more complex programs you are strongly advised to check the compatibilities manually in case your code uses version 2.0–specific features.

3.4 - Moving the Next Statement During Debugging

When stepping through your program one line at a time, you may need to jump a few lines back. To do this:

Right-click an arbitrary line

Choose Set Next Statement from the pop-up menu (see Figure 29).

This forces the debugger to jump to that line and continue debugging "normally" from there.

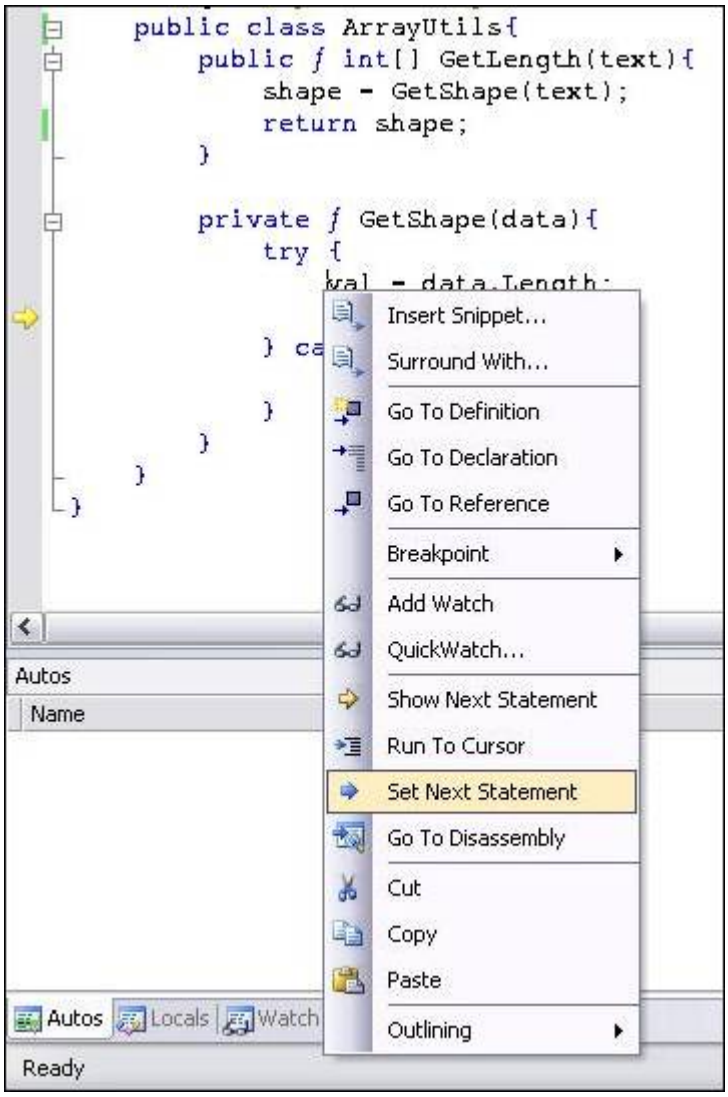


Figure 29. Set Next Statement

To jump back, and also jump forward in and out of control statements:

Drag the yellow arrow to any line.

Note: You cannot jump out of the current stack frame, so you are limited to moving inside your current method.

In addition, moving the current execution line can bring your program into states that under normal execution could not occur. Still, it's an extremely useful feature to rerun certain code lines without restarting your debugging session.

3.5 - Changing Variable Values in the Watch Window

In addition to moving the next-statement pointer, you can change variable values at debug-time. In the process of debugging your application, you may have moved your variables of interest into the Watch window (probably by dragging your variable there). The Watch window does more than display the current variable value and type; the value field is also editable.

For most value types this is accomplished by entering the new value.

Note: You need to change the internal tick value of DateTime variables.

As for reference types, you can re-reference variables to other variables. Let's say you have two instances of hash tables in your Watch window, named foo and bar. Setting the variable foo to the reference bar's hash table is as easy as typing bar in foo's value field. You can only change a reference variable to another reference variable of the same type (or its derived types).

Note: This can bring your program into states that under normal conditions would never be encountered.

3.6 - Executing SQL Procedures Through the Server Explorer

The SQL Server tree branch in the Server Explorer allows you inspect and analyze a SQL Server instance. In addition to the general features of inspecting a database table and Excel-like modifications of table contents by editing rows, the Server Explorer has other useful features.

VS.NET has limited capabilities of editing stored procedures. To view, edit, and modify stored procedures:

Right-click any stored procedure.

Choose Edit Stored Procedure from the pop-up menu.

Unfortunately, this feature does not compete well with the Enterprise Manager because error messages regarding syntax error are too general. Nevertheless, it's quite useful for its designed purpose of viewing, editing, and modifying stored procedures.

To execute stored procedures at design-time:

Right-click a stored procedure.

Choose Run Stored Procedure from the pop-up menu.

VS.NET inspects your stored procedure's parameter list. If necessary, the Run Stored Procedure dialog box is displayed:

Enter each parameter's value.

Execute your stored procedure and see the results.

3.7 - Customizing the Call Stack

A stack trace is a visual representation of the current hierarchy of method invocations as VS.NET steps through your program. While debugging your program, you step into methods and methods within methods. The stack trace keeps track of all these different levels.

To see the current stack trace:

Select Debug > Windows > Call Stack or

Press Ctrl-Alt-C,

Each method invocation is displayed on its own line, including the line-number and argument values. Each new method invocation is known as a stack frame.

The stack trace has been around in Visual Studio for a long time and is a widely known tool. The advantage of the stack trace window is that it allows you to identify how you get to the current execution point and also inspect the arguments that have been passed to the methods.

To make VS.NET immediately jump to the method invocation on a particular level of your program:

Double-click any line in the stack trace.

A relatively unknown aspect of the stack trace is that you can customize the Call Stack window. To do this:

Right-click the call stack.

Customize what appears there (see Figure 30) according to your requirements.

In addition, you can send the information regarding a single method invocation to a coworker:

Copy a stack frame to the Clipboard by pressing Ctrl-C.

To send your coworker the entire call stack:

Press Ctrl-A first, or

Before copying the selection to the Clipboard, Choose Select All from the context menu that appears after you right-click.

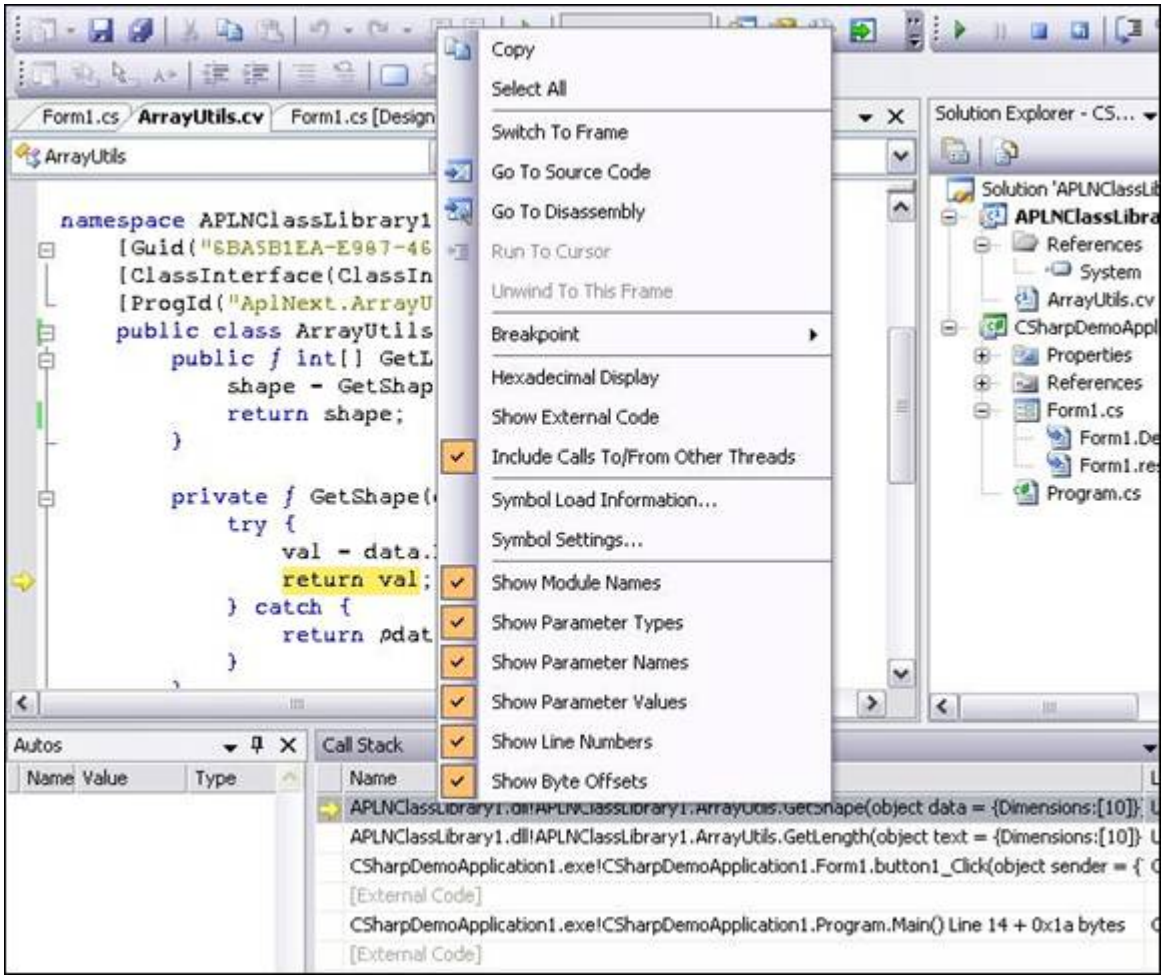


Figure 30. Customize the CallStack

3.8 - Attaching VS.NET to an Already Running Process

To instruct VS.NET to debug your program, you first are telling it to build your project (if necessary) then start the program in debug mode. This means that VS.NET is attached to the program so that it can react to breakpoints and other debug-related methods, assuming that the project was built with the debug release. To debug your program:

Press F5

There some cases, where you need, or want, to debug an already running process that has not been started with VS.NET you must:

Open the project for the program that is already running.

Select Debug > Attach to Process

A list of all active processes on your machine is displayed.

From the Processes dialog box, select the process you are interested in debugging and click Attach.

3.9 - Debugging Several Projects Inside the Solution

In a multi-project solution, VS.NET will start the project that you have marked as the "startup project." That project is indicated in the Solution Explorer with bold letters. If you start the other projects through Windows Explorer, you will see that VS.NET does not hit any breakpoints for those projects because VS.NET was not attached as a debugger to them.

It is possible to debug those programs anyway, using the instructions in, "Attaching VS.NET to an Already Running Process."

To instruct VS.NET to start a project and attachs itself to a specific program:

Right-click your project

Select Debug > Start New Instance from the pop-up menu.

You can repeat these steps several times to start multiple instances of your program and still debug them all. This is useful in debugging multi-threaded client-server scenarios.

Tell VS.NET which projects you want to start on each new debug session (see Figure 31):

Right-click your solution

Choose Set Startup Projects from the pop-up menu.

By default, VS.NET uses the Single Startup project, where only one project is started.

To start more than one project:

Switch to Multiple Startup Projects

Modify the Action value for each property: None, Start, or Start Without Debugging.

To control the order by which these multiple projects start:

Click the Move Up or Move Down button to position your projects in the list.

In a client-server scenario, you can use this to make sure that the server program is started before the client program.

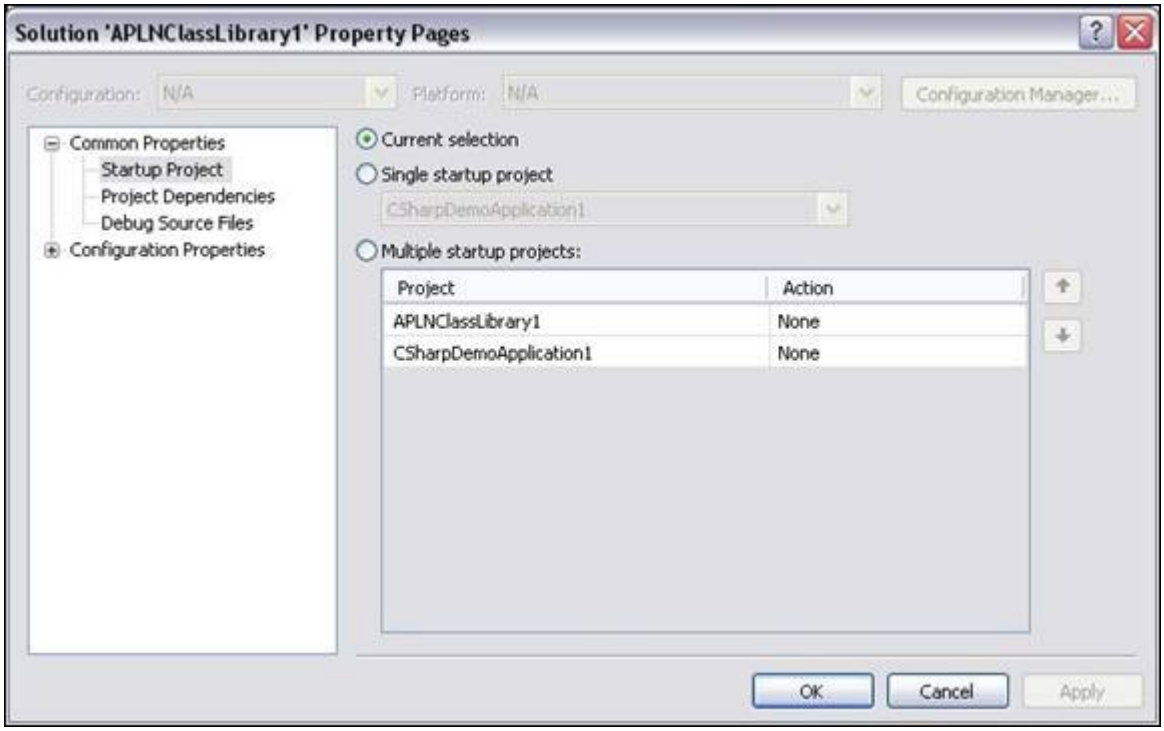


Figure 31. Multiple startup projects

3.10 - Breaking Only for Certain Exception Types

A good program usually catches all possible exceptions that can be thrown at runtime. However, this makes it a bit difficult for developers to debug a complex program that is still in development. Because there aren't any unhandled exceptions, VS.NET never catches an exception or prompts the user to break into the code whenever a specific exception is being thrown.

To specify the exceptions that developers are interested in defining, there is a setting in VS.NET. To utilize this setting:

Select Debug > Exceptions, or

Ctrl-Alt-E.

A tree view–style list of all possible exceptions that VS.NET can hook into (see Figure 32) will be displayed.

In addition to the many Common Language Runtime exceptions, you can hook into C++, Native Run-Time checks, and Win32 exceptions.

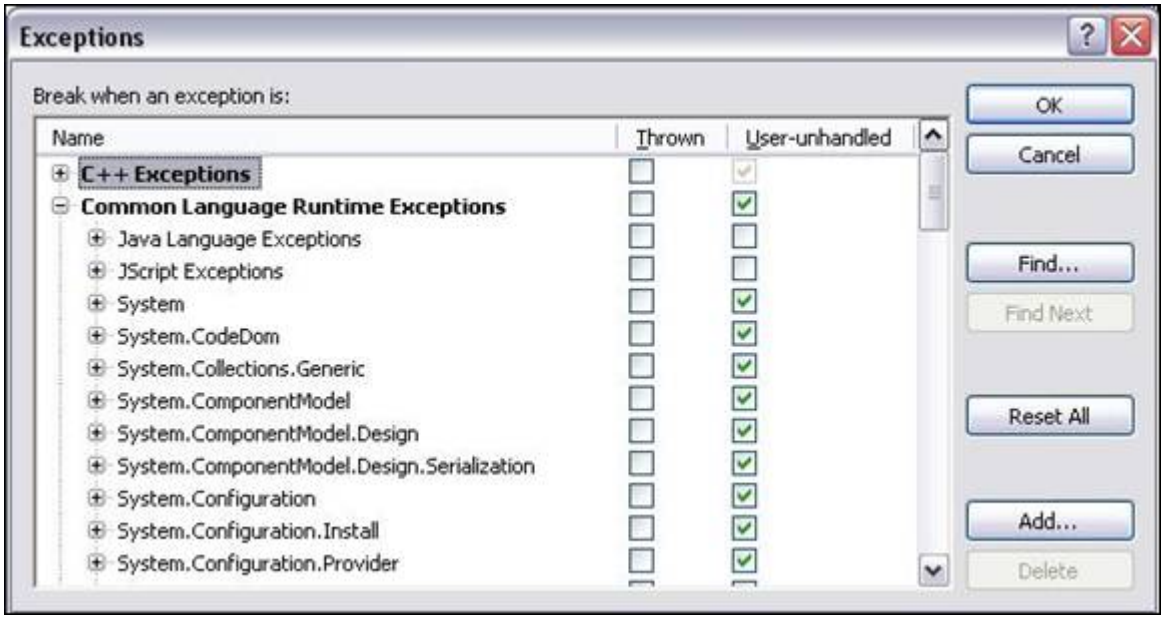


Figure 32. Break on specific exceptions

From this list you are able to:

Set, for each possible exception, exactly when to break into the debugger.

You can either hook into the debugger when a specific exception is thrown or when an exception is not handled. In the predefined .NET exceptions, you can hook into your own .NET exceptions.

To specify the complete, fully qualified string that defines your .NET exception, for example, "MyCompany.MyProduct.MyBusinessException":

Click the Add button in the Exceptions dialog box.

3.11 - Breaking Only When Certain Conditions Apply (Ctrl – Alt – B)

A heavily used method to add or remove exceptions is by clicking the gray vertical bar to the left of the editor. Clicking it adds and removes the red circle that indicates a breakpoint. By doing so, many developers never encounter the very useful conditions that you can set for breakpoints.
To access these conditions:

Set your breakpoint using your normal method.

Right-click your breakpoint.

From the context menu, choose Condition (Figure 33) to get to the Breakpoints window (Figure 34).

Two buttons stand out at the bottom of the Breakpoints window. To specify a condition under which a breakpoint becomes active:

Enter a .NET expression.

This can either be simply a variable name ("myBoolVariable") or a more complex .NET expression ("((System.DateTime.Now.Second % 10) == 0)"). You can choose to break into the debugger if the expression evaluates to True or when the expression value changes. Naturally, for the first option, the expression has to evaluate to a Boolean value. For the second option, your expression can be anything. VS.NET breaks into the debugger only if the runtime value of that expression changes from the last time it passes by this conditional exception (this implies that program execution has to pass by this code segment at least once previously, before it can recognize a change in value).

Given the flexibility of the expression, this feature can be very powerful. For instance, you can debug a snapshot of a DataSet only if the DataTable row size is greater than 0.

In VS.NET 2005, all the above-mentioned conditions are accessed in the following way:

Sset your breakpoint as you would normally do.

Right-click your breakpoint.

From the context menu, choose Condition to get to the same screen (see Figure 62).

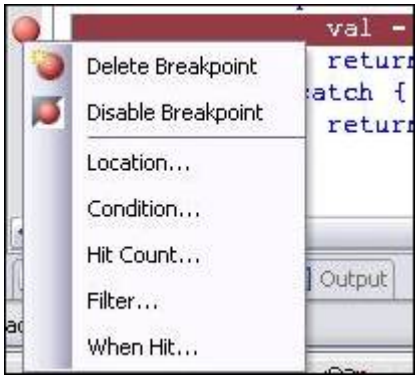


Figure 33. Set breakpoint condition.

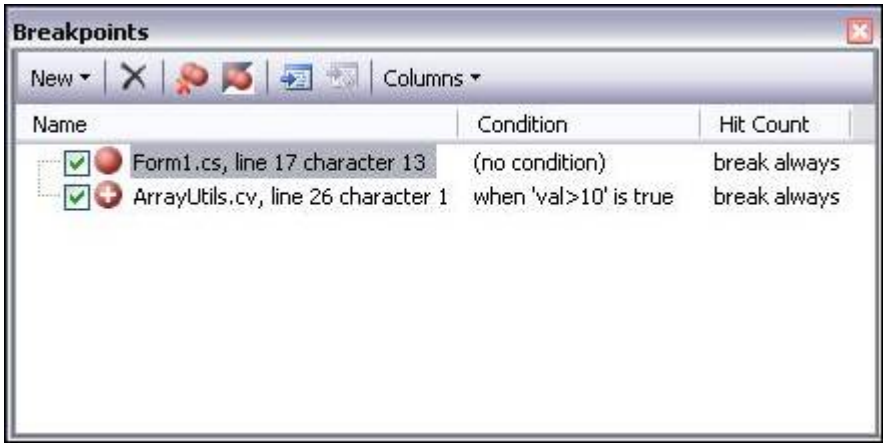


Figure 34. Breakpoints Window

To see and modify the condition in the Breakpoints window:

Open that window by selecting Debug > Windows > Breakpoints or

Pressing Ctrl-Alt-B.

A list of all breakpoints that you have set, along with their conditions will be displayed.

Note: You can disable breakpoints from this window as well, using the check boxes, or jump to their location in the file by double-clicking them.

3.12 - Saving Any Output Window

The Output window (Ctrl-Alt-O) shows a lot of trace information regarding your program execution. It lists whenever the .NET Framework loads a DLL for your application and, probably more importantly, all the messages that you have emitted with System.Debug.WriteLine.

To save all these trace logs:

> Press Ctrl-S to save the entire output to a file.

To search through the Output window:

> Press Ctrl-F

You can even apply some of the other editor tips and tricks such as Ctrl-C for copying an entire line or Ctrl-R, Ctrl-R for word-wrapping (although VS.NET 2005 now offers a button for word-wrapping in the Output window).

3.13 - Aligning UI Elements Automatically

If you are positioning UI elements in a Windows form, you have probably noticed various colored lines that appear on the form as you move or resize elements (see Figure 35). This allows you to snap your UI element to vertical or horizontal lines. Solid blue indicates lines to which other UI elements have already been snapped; they help you align elements consistently. Green dotted lines indicate the default margin between the UI element you are moving or resizing and the elements around it; they help you maintain uniform spacing between elements. Finally, solid red lines indicate that the text inside the current element is aligned with an adjacent UI element or its text.

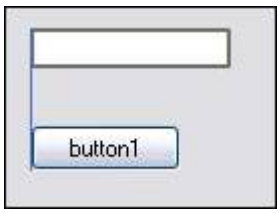


Figure 35. Align lines

To position UI elements without snapping to these colored lines:

> Press Alt to turn off automatic alignment temporarily.

To switch back to the grid where all UI elements are aligned to a predefined grid:

> Select Tools > Options > Windows Forms Designer > General and change LayoutMode back to SnapToGrid.

Note: After changing that value, you need to close and reopen the Designer view to use the newly selected layout mode. In SnapToGrid mode, you can press the Ctrl key to move elements without snapping them to the grid.

3.14 - Adding a Standard Menu Strip

Standard Windows applications use a common set of top-level menu items. In most cases, they are File, Edit, Tools, and Help. VS.NET 2005 allows you to add these default menu items to your own Windows forms applications. To add your own menu strip:

Drag a MenuStrip to your Windows form.

With the MenuStrip selected, the description panel below the Properties window shows an Insert Standard Items link (see Figure 74):

Click that link.

VS.NET inserts these standard items onto your MenuStrip. Menu items you insert contain the default submenu items as well. For instance, the File menu includes the usual New, Open, Save, Save As, Print, Print Preview, and Exit items, along with its default shortcuts, hot keys, and icons.

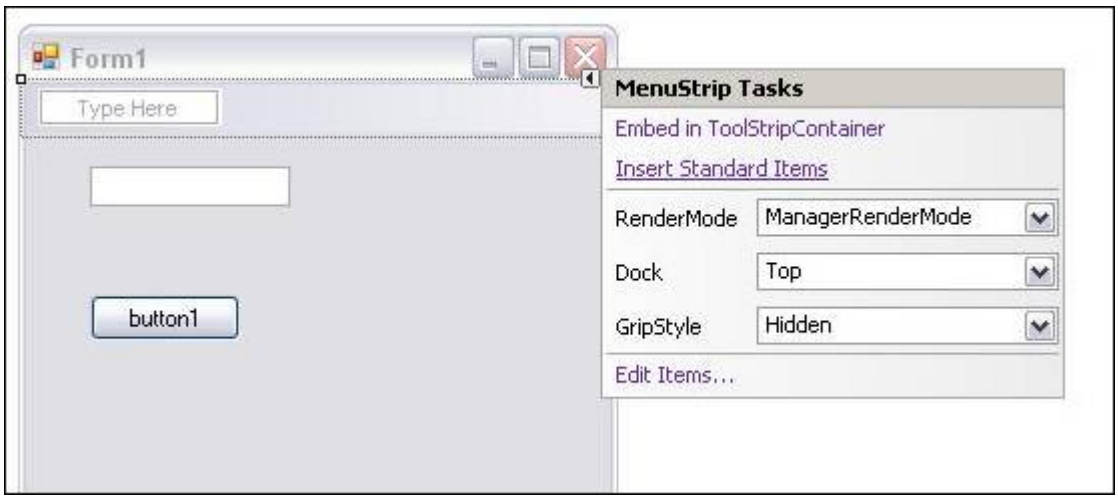


Figure 36. Menu strip standard items

3.15 - Setting the Tab Order of Controls

The tab order is the order by which controls on the form receive focus as you press the Tab key. You can control this order by setting the Tab Index property of each control to a number that corresponds to the position in this order. This can prove difficult at times because you don't know—and can't see—the other controls' tab index unless you select them.

Figure 37 - Tab Order button on the left of the Layout bar

VS.NET 2005 introduces a new way to set the tab order: the Tab Order button on the Layout bar (see Figure 37).

Click the Tab Order button to display the tab index for all UI elements on the form.

You now see all the tab indices.

Click repeatedly on each UI element to set the tab order in linear fashion.

The first element you select is given a tab index of zero. The next one you select has a tab index of one, and so on. As you set the index for each control, the background color of the tab index caption switches from blue to white, so you can keep track of which UI elements you have already tagged. To prevent you from accidentally selecting a wrong UI element, a gray rectangle surrounds the element you mouse over for better identification.

When you are done setting the tab order:

Click the Tab Order button again or

Press the Escape key.

3.16 - Importing and Exporting IDE Settings

VS.NET is an extremely powerful tool with many things in the IDE that you can customize to suit your specifications. Because you will become accustomed to your particular settings, moving from one machine to another can cause problems if you are not able to move your IDE settings along with you.

VS.NET 2005 allows you to export your IDE settings to an XML file (the extension is actually ".vssettings"). To import it into another instance of VS.NET on another computer:

Select Tools > Import > Export Settings.

In the tree view shown in the Import/Export Settings dialog box, you are presented with all the customizable options you can export (see Figure 38).

Check the options you want to be part of your profile.

Export them to the .vssettings file.

Figure 38. Export VS Settings

Import the .vssettings file to another VS.NET IDE.

Select which settings you want to import and which ones to ignore.

In the same dialog box you can also reset your complete VS.NET IDE to a particular profile. These might be custom profiles that you saved before. You can also reset to the default installation settings (which is just another regular .vssettings file).

To create a master .vssettings file for coordination between co-workers, e-mail it to all your team members so that they can import it individually. You can also create the single .vssettings file and place it on a well-known network share on the intranet. To obtain these settings have the members of your team:

Select Tools > Options > Environment > Import > Export Settings > Team Settings.

There they have to turn on Track Team Settings File and point it to that shared .vssettings file. Next time they start their IDE, it will detect the file and import it. One advantage of this feature is that another trusted team lead can export a version of the shared .vssettings file and overwrite it, so that the IDEs of each developer will detect that change and import it upon the next startup.

3.17 - Closing All Other Windows

It's very common to have a lot of files open at the same time when developing your program. After working for a while, you might have several dozen files open and want to close all of them except the one on which you are currently working.

To close all the open files:

Right-click one of the file tabs.

Choose Close All But This from the pop-up menu.

This option does exactly what it says (see Figure 39).

Other menu options new to VS.NET 2005:

1. Open Containing Folder
   -starts up Windows Explorer and opens the folder in which your file is located.

2. Copy Full Path
   -copies the full file path of the selected file into the Clipboard.

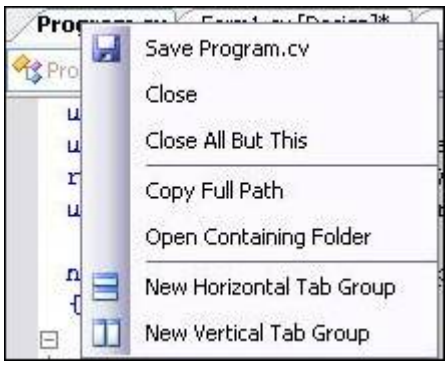


Figure 39. File tabs options

3.18 - Showing Shortcuts for All Buttons

Using and memorizing shortcuts whenever available, gives you a strong advantage when developing. It naturally increases your speed and therefore your efficiency. Keyboard shortcuts prove to be faster than manipulating the mouse. Many VS.NET menu and submenu items have these shortcuts which are seen every time you click the menu item.

This reminder is also available for the toolbar buttons. To see this reminder:

Select Tools > Customize

Check both the Show ScreenTips on Toolbars and Show Shortcut Keys in ScreenTips options.

Now as you mouse over a button, the ToolTip that appears after a small delay will also show the button's keyboard shortcut, if available.