VisualAPL – Application-shared Datastore

VisualAPL implements the "Application-shared Datastore" so that it is easy, efficient and safe to share data within an application system across classes.

Before delving into the VisualAPL Application-shared Datastore, here is some background information on “global” variables in .Net.

When building a .Net application system each component is implemented within a class and that class is encapsulated so that only the public members, e.g. methods, properties and events that have the public attribute, are available for access by other classes.

This application system structure means that methods need to have explicit arguments and results and data is shared between classes only by passing via the values of these arguments and results. Sometimes classes use properties instead of methods, but these are equivalent since properties are designed with explicit arguments (for properties whose values can be set) and results (for properties whose value can be gotten).

Programmers using legacy APL language implementations have employed explicit arguments and results, just as described above, but they have also used variables and data which are global (or implicit).

Because .Net programming languages adhere to OOP (object-oriented programming) principles, the concept of global or explicit data and variables is not directly implemented. In fact, this restriction on global variables implicitly shared among classes is often the subject of vitriolic debate among .Net programmers. Here are a few samples of that kind of ‘debate’:

- http://bytes.com/groups/net-c/226096-global-variables
- http://www.vbforums.com/showthread.php?p=1287485

Note that C# does implement a keyword global, but it is not used to specify a global variable. See here for the use of this key word in C# to prevent namespace conflicts: http://msdn.microsoft.com/en-us/library/cc713620.aspx

Note that VB.Net does implement a keyword global, but the ‘static’ property implementation described subsequently in this document, underlies implementation.

Just what are the ‘bad’ things about designing an application system with a global variable? Not surprisingly, the ‘dangers’ of global variables are exactly the same in .Net languages and in legacy APL language implementations:

- Potential conflicting variable names is the essential “bad” thing about global variables. For example, two different classes, or APL functions, create a global variable with the same name but for different purposes.
- Potential lack of “type safety”. For example, two different classes attempt to modify a strongly-typed global variable with values which are of a different data type.
- Potential conflicting dependencies among classes. For example, a class is designed which uses the value of a global variable, but when that class is used in a different application system without the global variable present, the functionality fails.

In .Net languages there are several ways to indirectly implement global variables:

- Adding a variable to the “Application” class which exists in every .Net project and is available within any class used by the same thread as that project. A good example of this is the start-up parameters list in the Windows short-cut for an .exe, which once captured can b e made available to any class used by that application system. Here is an example of that type of C# code:

```
//Establish get/set methods for the application system's data class, so
it is modifiable throughout the application worksession's lifetime

private App_CmdLineArgs _ACLA = new App_CmdLineArgs();
public App_CmdLineArgs ACLA
{
   get { return _ACLA; }
   set { _ACLA = value; }
}

private void Application_Startup(object sender, StartupEventArgs e)
{
  ((App)Application.Current).ACLA.Stringvec = e.Args;
}
```

- Creating a class with a “static” property. A property may have the static keyword added so that, even if no instance of the class exists, the value of the property is the same fixed value for any class which references that static property. Essentially the data has been encapsulated into the class is thus shared by any other class. The class name instead of the class instance name is use to reference a static property, e.g. “ClassName.StaticVariableName”. See here for more information:

http://msdn.microsoft.com/en-us/library/w86s7x04(VS.71).aspx

http://www.c-sharpcorner.com/UploadFile/rajeshvs/PropertiesInCS11122005001040AM/PropertiesInCS.aspx

- An assembly can specify a non-mutual, intransitive 'trust relationship' so that the trusted assembly may access all the non-public types of the trusting assembly.
    - The trusting assembly must reference the System.Runtime.ComplierServices namespace
    - The trusting assembly must include the attribute [assembly: InternalsVisibleTo("AssemblyXPublicKeyToken=…")]
- Using explicit arguments and results for all methods in a class, i.e. avoiding global variables entirely.

Finally we can get back to discussing the VisualAPL “Application-shared Datastore” which provides an additional method for sharing of any object between classes. This is done in a “safe” way by requiring that any class that uses a global object must declare that global use within the class in which the sharing will occur: svglobal My_Variable

One can immediately see how easy creating a global variable is in VisualAPL compared to other methods previously described in this document.

Once the object has been added to the Application-shared Datastore, any other namespace, class or project in the Visual Studio solution will be able to reference the object using the same syntax.

Objects which have the svglobal attribute are global with respect to all functions within the class and only need to be defined using svglobal once in that class.

VisualAPL precisely defines the order of name resolution for objects in the Application-shared Datastore. In any class when an object is referenced, VisualAPL “looks for” local variables with the same name first, then global variables (within the class) with the same name and finally svglobal objects with the same name.

There are some additional VisualAPL □-functions available to manage the Application-shared Datastore:

- `□svc “My_Obj”` returns the changed status [0/Not changed 1/changed] of the “My_Obj” entry in the Application-shared Datastore since the local class’ last reference or assignment of that entry was performed.
- `□svs “My_Obj”` returns the shared status [0/Not shared 1/shared] of the “My_Obj” entry in the Application-shared Datastore.
- `□svd “My_Obj”` deletes the “My_Obj” entry from the Application-shared Datastore.
- `”EH_FN”□svget “My_Obj”` subscribes to the event that the “My_Obj” entry in the Application-shared Datastore has been referenced in which case the “EH_FN” event handler VisualAPL function will be executed.

  The required function signature of the “EH_FN” event handler function is:
  - Function EH_FN(name, value)
  - “name” is the (string) name of the entry, e.g. “My_Obj”
  - “value” is the value of the entry

- `”EH_FN”⎕svset “My_Obj”` subscribes to the event that the “My_Obj” entry in the Application-shared Datastore has been assigned in which case the “EH_FN” event handler VisualAPL function will be executed.

  The required function signature of the “EH_FN” event handler function is:
  - Function EH_FN(name, oldvalue, newvalue)
  - “name” is the (string) name of the entry, e.g. “My_Obj”
  - “oldvalue” is the value of the entry before assignment
  - “newvalue” is the value of the entry after assignment