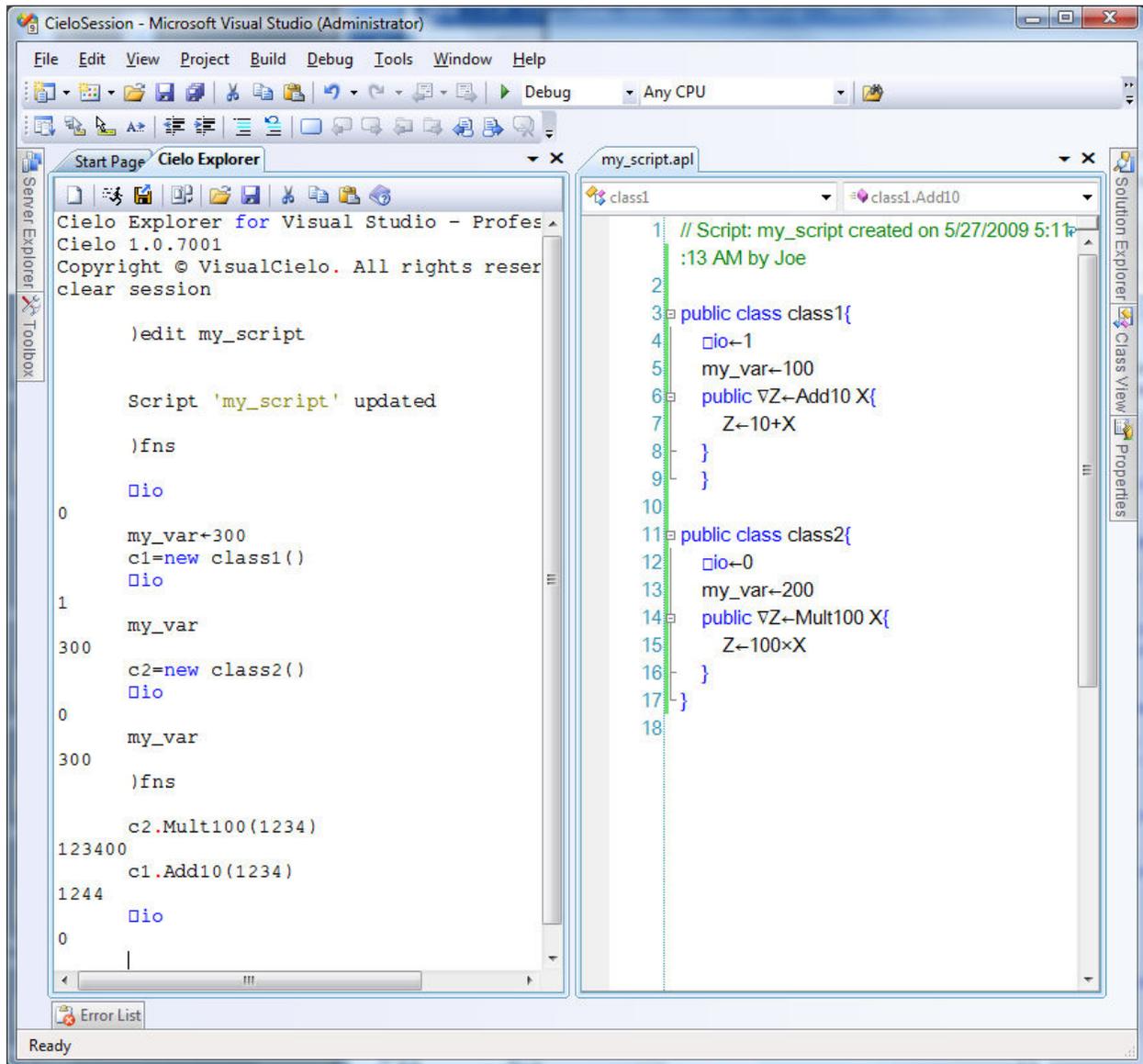


Here is an example of using classes in a Cielo Explorer script to isolate the values of user-defined variables and function, e.g. my\_var and Add10 and Mult100 and also variables in the system 'state', e.g. `!io`.



Notice that the system 'state' variables, e.g. `!io`, etc., are established in the session when the class instance is created.

Another system 'state', or 'context' is also controllable when using the VisualAPL primitive execute operator. Refer to the documentation of the primitive execute operator in the file "...\ApiNext\VisualAPLProfessional\Documentation\Visual\_APL\_Help.chm" for details:

The screenshot shows the Visual APL Programmer's Reference help window. The left pane displays a tree view of the documentation structure, with 'Operators' expanded and 'Execute' selected. The right pane shows the 'Execute' operator documentation, including its description, syntax, where clause, remarks, and advanced dynamic execution features.

**Visual APL Programmer's Reference**  
**Execute**

Executes the code supplied by *expr1*

```
result ← execute 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 = execute "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 *ws1*. The field *ws* contains all static data which exists in the current context of where you reference *ws*, and *ws1* 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 *ws1*.

By default, when you run a dynamic execution and do not specify the global context in which it will run, the *ws1* (or *ws* for static methods) is passed as the default global dictionary.

**Dynamically defining contexts:**