```
clearlog();
/*
An effective JSL architecture to develop, test, and
maintain applications in JMP software

Peter Wiebe, PhD.; Mark Anawis; Kevin True

Applications developed in JMP are powerful, deployable, and
maintainable solutions for complex analytical processes.
Here we describe a JSL application architecture that
manages namespace, variable naming conventions, and memory
usage. By effectively combining the JSL commands 'include',
'function', and 'expr' as the architecture of the JSL Add-
In, and leveraging the Add-In path
"$ADDIN_HOME(Unique_ID)\file.jsl", an otherwise complex and
lengthy JSL script can be broken into several smaller JSL
scripts. Each of these smaller scripts can be developed
with their own variable naming conventions, tested
independently or in combination with the other application
scripts, creating a more manageable development process for
a JSL programmer. Exercising the 'new context' argument
available within the 'include' command effectively manages
potential namespace collisions, while the 'default local'
argument available within the 'function' command
conveniently manages variable memory space. In addition,
creating an application namespace allows for information to
be passed across the application as necessary, while
otherwise minimizing memory space. The application
architecture described is more convenient for a JSL
programmer to develop, test, and maintain JMP applications.
```

Applications developed in JMP are powerful, deployable, and maintainable solutions for complex analytical processes.

As the JMP scripting language (JSL) has evolved with new versions of JMP, the capability to create custom applications has become more convenient. Indeed, complex analytical processes can be customized for industry specific usage. However, as more complex JMP applications are created, there is an increased burden on the software developer to develop a solution that is robust and maintainable.

As the JSL script for an application grows, the software developer faces new kinds of challenges developing and troubleshooting. For example, when a script exceeds a thousand lines, merely scrolling through the code to troubleshoot becomes prohibitive. Similarly, as the number of variables grows, managing naming conflicts or recalling variable names when needed becomes prohibitive. Here we describe a JSL application architecture that manages namespace, variable naming conventions, and memory usage.

By effectively combining the JSL commands 'include', 'function', and 'expr' as the architecture of the JSL Add-In, and leveraging the Add-In path "$ADDIN_HOME(Unique_ID)\file.jsl", an otherwise complex and lengthy JSL script can be broken into several smaller JSL scripts.

The Include function is useful to begin dividing one script into several manageable script pieces. In this document I will refer to these pieces as child scripts whereas the script that includes the Include function will be referred to as the parent script. This is a helpful terminology to illustrate the relationship between scripts. According to the JMP online documentation, the Include function opens a script file, parses the script in it, and executes the JSL in the specified file.

http://www.jmp.com/support/help/Advanced_Programming_Concepts.shtml#306193

Here is its basic syntax and an example:

```
include("pathname");*/


include( "$SAMPLE_SCRIPTS/chaosGame.jsl" );

/*
An important quality to note is that the pathname can be
fixed or variable. In the previous example, the variable
path name "$SAMPLE_SCRIPTS" was used. Path variables are
described in the JMP online documentation:

http://www.jmp.com/support/help/Path_Variables.shtml


The Include function example provided above can be
rewritten using a fixed path; however, while it will work
in my JMP software at the time of writing this document, it
may not work in other versions of JMP or on other people's
computers. Here is an example:
*/
include( "/C:/Program
Files/SAS/JMP/10/Samples/Scripts/chaosGame.jsl" );

/*
As a JSL application developer, to leverage the power of
the Include function effectively, there needs to be a
variable path name that can be used in an AddIn that will
reliably point to the application script files. At the time
of preparing this document, a notably absent variable path
name from the JMP online documentation for path variables
is the following:

"$ADDIN_HOME(Unique_ID)/file.jsl"

However, it is described elsewhere in the JMP online
documentation:

http://www.jmp.com/support/help/Utility_Functions.shtml


This path variable allows the Include function to be more
powerful for a JSL application developer. Scripts can be
```

```
attached to an AddIn while it is constructed, and applied
using the following construct.*/

     include( "$ADDIN_HOME(Unique_ID)/file.jsl" );

/*The Include function and perhaps even the AddIn_Home path
variable may be familiar to some developers. Together they
are a reliable architecture to divide a script into several
child scripts.

To begin leveraging this architecture, we are going to
create a working directory and register a new JMP AddIn
directing it to that directory.*/

     createdirectory("$DOCUMENTS/discovery2014");
     registeraddin(
          "abbott.wiebe.disc2014",
          "$DOCUMENTS/discovery2014"
     );

/*Now that we have a working directory, let's create and
save a "Hello World" test child script. This process can
also be performed by creating and saving a JSL file with
the JMP user interface.*/

     savetextfile(
     "$ADDIN_HOME(abbott.wiebe.disc2014)/HelloWorld.jsl",
          char(
               expr(
                    var1="Hello";
                    var2="World";
                    print(var1||" "||var2)
               )
          )
     );

/*So, we now have a working directory and test child script
saved to the location. Let's exercise our architecture by
running an Include function in our parent script that
executes the child script.*/

     clearlog();
     deletesymbols();
```

```
    include(
"$ADDIN_HOME(abbott.wiebe.disc2014)/HelloWorld.jsl");
    showsymbols();

/*Review of the JMP log reveals the script was executed as
expected.*/


    "Hello World"
    // Global

    var1 = "Hello";
    var2 = "World";

    // 2 Global
```

/*Exercising the 'new context' argument available within the 'include' command effectively manages potential namespace collisions and smaller scripts can be developed with their own variable naming conventions.

When variables are defined, if there is no explicit syntax to manage the variable namespace, then the variables are in the global namespace. This means that any variable defined within a JMP session can be conveniently accessed and modified through any script window or application. This can result in accidental namespace collisions. For example, two JMP scripts may be executed in a session where one redefines a variable that is in use by the other JMP script. This is typically undesired, and can cause an application to not function as expected, or not at all.

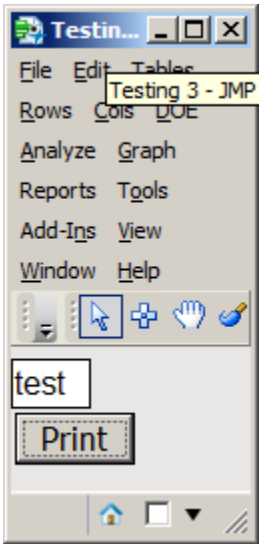
There are several options to manage namespace, which will not be discussed here at length. Instead we will describe an implementation that simplifies namespace management. Here is one notable function*/:

```
    namesdefaulttohere(1);

/*Namesdefaulttohere is used to create a new namespace
context within the script. This can be very powerful to
encapsulate a namespace to protect it from unexpected
namespace collisions. Once this namespace property is
```

```
established in a script, the namespace management can be
extended to other functions.*/

     namesdefaulttohere(1);
     clearlog();
     deletesymbols();
     var1="Goodbye";
     include(
"$ADDIN_HOME(abbott.wiebe.disc2014)/HelloWorld.jsl",
          <<namesdefaulttohere(1)
     );
     showsymbols();


/*Note that in this example, not only is namesdefaulttohere
at the beginning of the script, it is also added as an
argument to the include function. In addition, the variable
var1 has been set to "Goodbye" prior to executing the
include function. Review of the JMP log:*/

     "Hello World"
     // Here

     var1 = "Hello";
     var2 = "World";

     // 2 Here

/*Although both the parent and child script both have the
namesdefaulttohere, var1 defined in the child script
replaces var1 defined in the parent script. This can be
controlled from the parent script. The Include function has
a named option, New Context, which creates a namespace that
the included script runs in as long as a namespace has been
previously defined and is not the global namespace. This
namespace is an anonymous namespace and it is independent
from the parent script's namespace. The syntax would be as
follows:*/

     namesdefaulttohere(1);
     clearlog();
     deletesymbols();
     var1="Goodbye";
```

```
	include(
"$ADDIN_HOME(abbott.wiebe.disc2014)/HelloWorld.jsl",
		<<namesdefaulttohere(1),
		<<New Context
	);
	showsymbols();

/*Review of the JMP log:*/


	"Hello World"
	// Here

	var1 = "Goodbye";

	// 1 Here

/*The result is that var1 in the parent script remains
unchanged as "Goodbye", and var1 in the child script is set
to "Hello", which is shown by the "Hello World" message in
the log from the expression.

As a result, all of the variables and the naming convention
of each child script are encapsulated into an independent
namespace from all other scripts as well as from multiple
executions of the same child script. If for example, the
same Include syntax was listed two or more times in another
script, each execution of the Include syntax would be
independent. Consider the following example:*/

	savetextfile(
	"$ADDIN_HOME(abbott.wiebe.disc2014)/WindowExpr.jsl",
		char(
			expr(
				Newwindow("Testing",
					vlistbox(
						tbox=texteditbox("hello"),
						buttonbox("Print",
							<<setscript(
								var1=tbox<<gettext();
								print(var1)
							)
						)
					)
```

```
                    )
                )
            )
        )

/*In this expression, a new window is created with a text
edit box that allows a user to type in a string. When the
user clicks on the button box labelled "Print", the value
in the text box is sent to the JMP log.*/

    namesdefaulttohere(1);
    clearlog();
    deletesymbols();
    var1="Goodbye";
    include(
"$ADDIN_HOME(abbott.wiebe.disc2014)/WindowExpr.jsl",
        <<namesdefaulttohere(1),
        <<New Context
    );
    showsymbols();


/*The inclusion of the New Context argument encapsulates
the JMP script from each new window created from each other
and from the parent script. Execute the code three times,
and review of the JMP log after changing one of the boxes
to "test":*/
```



```
	// Here

	var1 = "Goodbye";

	// 1 Here

	"test"
```

/***Leveraging the Function function with default local***

Another very useful function in JSL is the function Function. The JMP online documentation provides a detailed explanation.

http://www.jmp.com/support/help/Advanced_Programming_Concepts.shtml#306193

Let's create a modified version of the HelloWorld.jsl child script so that it is a function that accepts two strings, and prints them to the log. Here is the function:*/

```
	function({var1, var2},
		{Default Local},
		//define intermediate variable
		concat_var = var1||" "||var2;
		//print it to the log
		print(concat_var);
		//output the evaluation to the function object
		concat_var
	);
```

/*Here is a script to create it as a child script in our working directory.*/

```
	savetextfile(
	"$ADDIN_HOME(abbott.wiebe.disc2014)/HelloWorldfnc.jsl",
		char(
			expr(
				function({var1, var2},
					{Default Local},
```

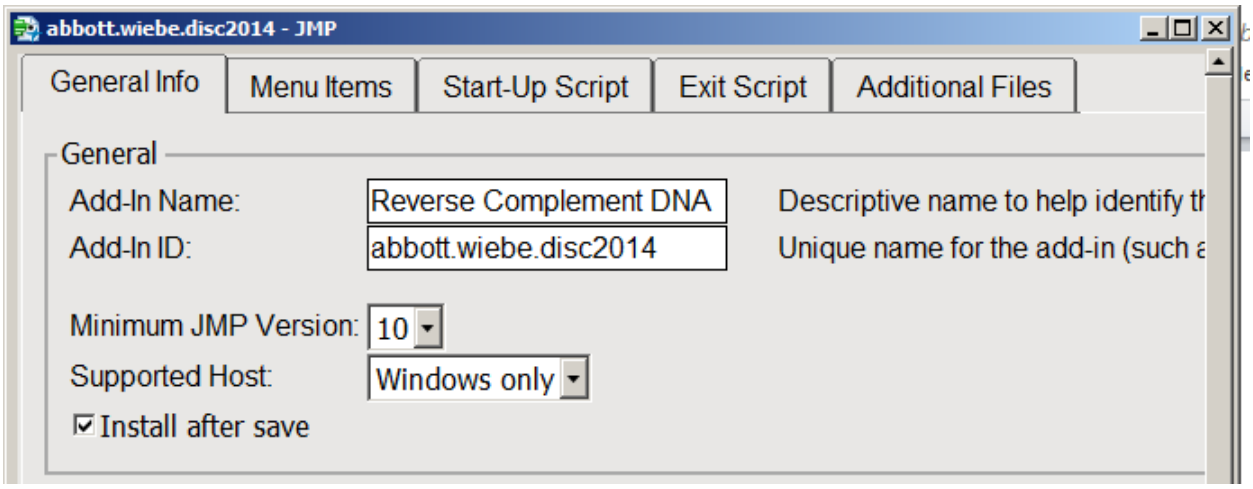
```
                    concat_var = var1||" "||var2;
                    print(concat_var);
                    concat_var
                );
            )
        )
    );

/*Now instead of the child script containing two strings
"Hello" and "World", we can pass them into a function from
the parent script. This is a two-step process. First the
Include function is executed to define the function, and
then the function is executed, passing the two strings
defined in the parent script.*/

    namesdefaulttohere(1);
    clearlog();
    deletesymbols();
    concat_var = "Goodbye World";
    MyFunction = include(
    "$ADDIN_HOME(abbott.wiebe.disc2014)/HelloWorldfnc.jsl",
    );
    FunctionOutput=MyFunction("Hello","World");
    showsymbols();

/*Review of the JMP log:*/

    "Hello World"
    // Here

    concat_var = "Goodbye World";
    FunctionOutput = "Hello World";
    MyFunction = Function( {var1, var2},
        {Default Local},
        concat_var = var1 || " " || var2;
        Print( concat_var );
        concat_var;
    );

    // 3 Here

/*The parent script defines the function name, controls
encapsulation, and is the source for the variables passed
```

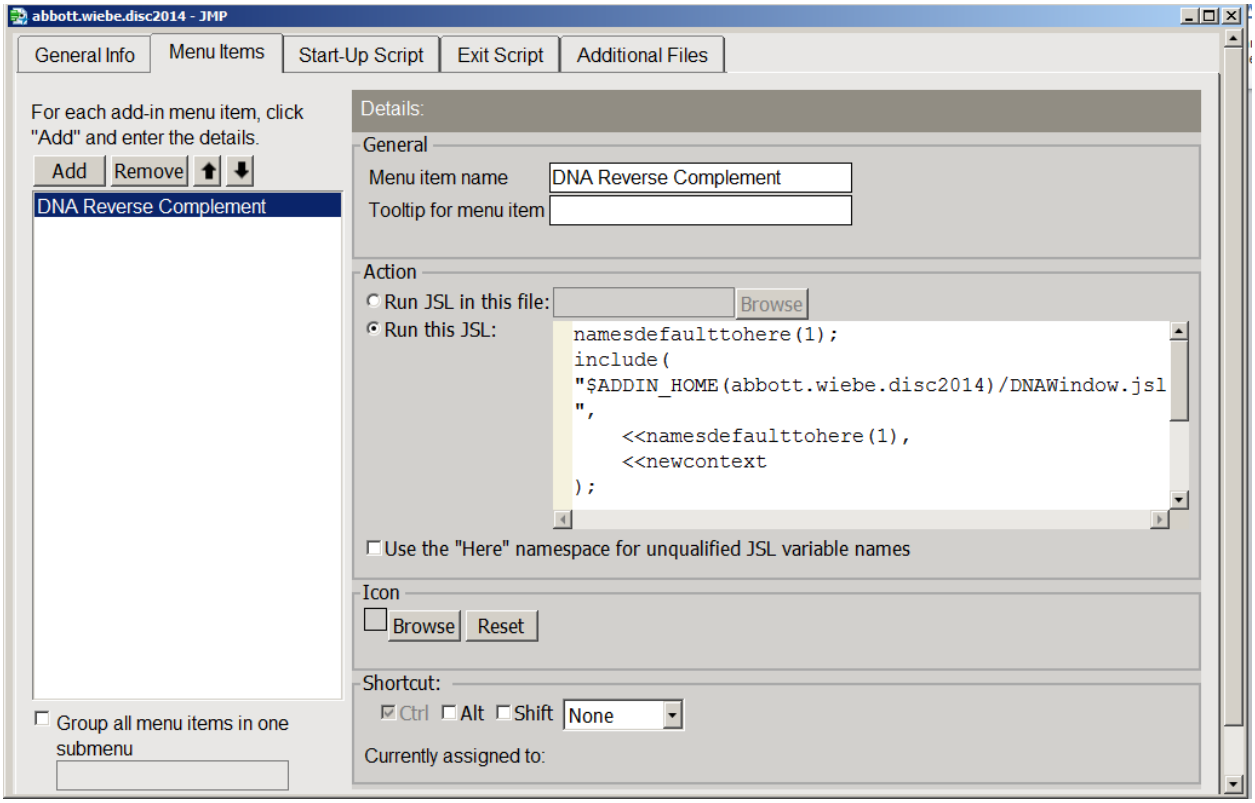
to the function. Should the function return an output, a variable object can be defined. In this example, 'FunctionOutput' is the object variable that references the output of the function. The function syntax and variable namespace is all determined in the child script. Once the function completes its execution, the memory space for the function is released back to the system, since the Default Local option is included.

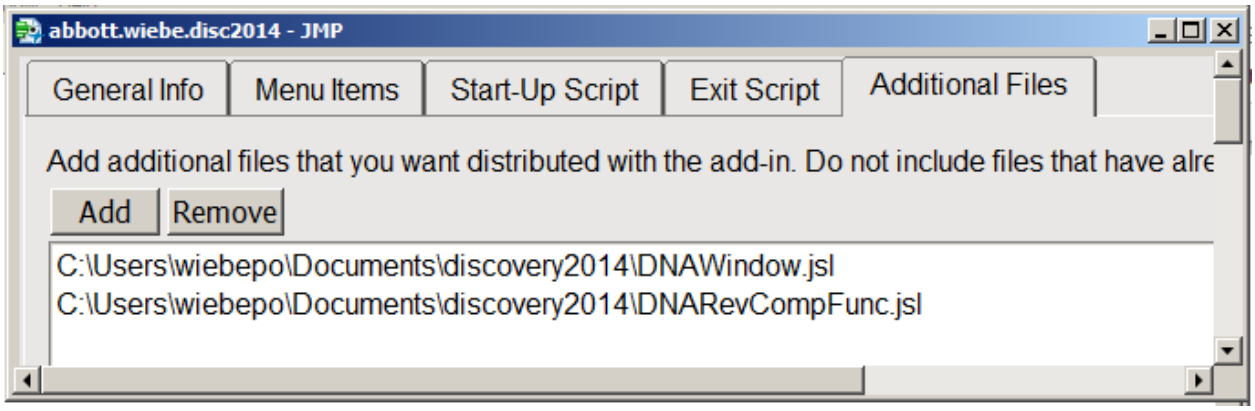
Let's bring all the elements together in an example. We will create a JMP AddIn that will provide a reverse complement DNA sequence. This will be accomplished by creating one child script that is a function to perform the reverse complement, and another to provide the user interface. The function accepts a string composed of A, T, C, or G, and returns a string that is the reverse complement:*/

```
Save Text File(
"$ADDIN_HOME(abbott.wiebe.disc2014)/DNARevCompFunc.jsl"
,
     Char(Expr(
     Function( {DNAString},
          {default local},
          Concat Items(
               Reverse(
                    {"T", "A", "G", "C"}[
                         Loc(
                         Design(
                              Words( DNAString, "" ),
                              {"A", "T", "C", "G"}
                         )
                         ) -
                         ((0 :: Length( DNAString ) - 1)
                         * 4)`

                         ]
               ),
```

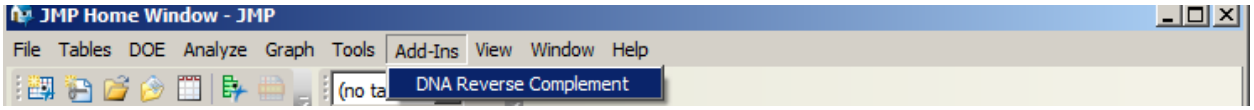
```
                    ""
               )
          )
          ))
     );


     /*Here is the script to create the child script for the
     user interface*/


     Save Text File(
     "$ADDIN_HOME(abbott.wiebe.disc2014)/DNAWindow.jsl",
     Char(Expr(
     RevComp=include(
     "$ADDIN_HOME(abbott.wiebe.disc2014)/DNARevCompFunc.jsl"
     );
     newwindow("DNA",
          vlistbox(
               tbox=texteditbox("Enter DNA Sequence",
                    <<setscript(
                         NewSeq=RevComp(tbox<<gettext());
                         nbox<<settext(NewSeq);
                    )
               ),
               nbox=textbox("")
          )
     )
     )))

/*Finally, let's create the JMP AddIn through the user
interface, manually configuring the files and adding the
following to the interface.*/


     namesdefaulttohere(1);
     include(
     "$ADDIN_HOME(abbott.wiebe.disc2014)/DNAWindow.jsl",
          <<namesdefaulttohere(1),
          <<newcontext
     );
```

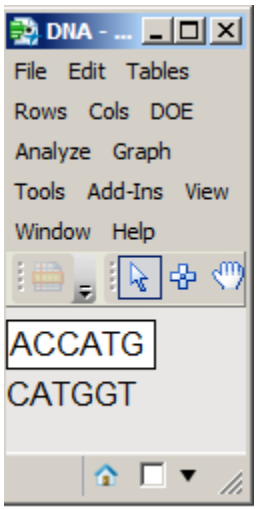

abbott.wiebe.disc2014 - JMP
General Info
Menu Items
Start-Up Script
Exit Script
Additional Files
General
Add-In Name:
Reverse Complement DNA
Descriptive name to help identify th
Add-In ID:
abbott.wiebe.disc2014
Unique name for the add-in (such a
Minimum JMP Version:
10
Supported Host:
Windows only
Install after save


abbott.wiebe.disc2014 - JMP
General Info
Menu Items
Start-Up Script
Exit Script
Additional Files
For each add-in menu item, click "Add" and enter the details.
Add
Remove
DNA Reverse Complement
Group all menu items in one submenu
Details:
General
Menu item name
DNA Reverse Complement
Tooltip for menu item
Action
Run JSL in this file:
Browse
Run this JSL:
namesdefaulttohere(1);
include(
"$ADDIN_HOME(abbott.wiebe.disc2014)/DNAWindow.jsl
",
<<namesdefaulttohere(1),
<<newcontext
);
Use the "Here" namespace for unqualified JSL variable names
Icon
Browse
Reset
Shortcut:
Ctrl
Alt
Shift
None
Currently assigned to:




```
/*Save the file, and select it from the Add-Ins menu*/
```





Conclusion

By exercising a few functions and their built in arguments, an otherwise large and complex JMP script can be broken into manageable pieces. The smaller child scripts can be controlled through a parent script, and the child scripts can be encapsulated to manage both memory and namespace. Leveraging these scripting techniques can result in a more manageable custom application design.*/