DYALOG APL

Simple APL Library Toolkit (SALT)

For Version 1.54 dated 2009-08-12

Introduction

SALT is a source code management system for Classes and script-based Namespaces in Dyalog APL. SALT is also capable of storing code (e.g. functions) but it was designed for scripted namespaces. The source code for each object (class or namespace) is stored in a single Unicode text file with a default file extension of ".dyalog". SALT also supports the loading and starting of applications from an "application file" with an extension of ".dyapp".

Classes provide a convenient mechanism for wrapping tools in a way which makes them easy to share. SALT is intended to provide a common mechanism for APL users to develop and share code in "Open Source" libraries.

SALT aims to provide the minimum useful set of functionality for a small team of developers, and provides the following set of functions (in a "likely" order of usage):

Function	Modifiers	Discussion
List '*folder|object*'	-Folders	Only list folders, not individual objects
	-Recursive	Recursively list contents below named folder
	-Versions	Also list intermediate version copies of object
	-Raw	Returns unformatted data
	-Full=1 or -Full	Show full pathnames below root folder
	-Full=2	Show complete pathnames
Explore '*name*'		Opens *Disk Explorer* if name is a folder or *Notepad* on source if name is an object
	-Mode=Normal|Max	in Normal or Maximized window
	-Use=	Use specific program to edit the file
	-Permanent	Remember permanently the program to use
Load '*name*'	-Version=	Loads a particular version
	-Target=Namespace	Defines the object in a specific ns
	-Source	Returns the source as a nested vector instead of defining the object in the workspace
	-NoName	Returns a ref to the object but does not "name" it in ws
	-NoLink	Do not manage the source for the object after loading it into the workspace
	-Disperse[=nl]	Brings in the objects specified (or all)
New '*name*' *[arg]*		Creates an instance of class 'name'
		without naming the class in the workspace
	-Version=	Use a specific version
Save '*ref file*'	-Version=	Save particular version number
	-Noprompt	Skip prompt to replace the file
	-MakeDir	Create folders if necessary
	-Convert	Convert the namespace to script form

Compare ‘*name’*	-Version=n	Compare current (last) version with version n (default is to compare last 2 versions) or workspace if n=’ws'
	-Version=n1 n2	Compare two particular versions
	-Use=	Use specific program to compare (APL default)
	-Permanent	Remember permanently the program to use
if *Use=APL* is used	-Zone=	# of lines to show before and after matches
	-Trim	Ignore spaces at the ends
RemoveVersions	-Version=[<>]n	Drop specific version(s) n
	-Collapse	keep last version
‘*name’*	-All	Forget ALL backup versions but the last one
Boot ‘*app’*		Loads and runs an application from a .dyapp file or from a function in a .dyalog file
	-Function	
	-Xload	Do not run the []LX if any
Settings'[*id [v]]*'		Returns[/sets] registry values
	-Reset	Reload settings from registry
	-Permanent	Save settings to registry
Snap ‘[*path*]’		Save new and modified objects to path given or current *workdir* if unspecified (see down below)
	-Version[=x]	Use version number (=x)
	-NoPrompt	Skip replacement prompt
	-MakeDir	Create intermediate folders if necessary
	-Show[=details]	Only show, do not save. Details are filename
	-FilePrefix=str	Prefix all files by ‘str’
	-Class=[~]3 4 9	Use (~=don’t) classes given
	-Pattern=[~]str*	Use (~=don’t) names starting with ‘str’
	-LoadFn[=newp]	Save boot function <merge_ws> in *path* or *newp*
	-Vars[=vn]	Save #’s variables in a separate namespace *vn* for when booting the workspace
	-Convert	Convert namespaces permanently into source form

The use of text files as a storage mechanism means that SALT and other tools written in APL can be combined with industry standard tools for source code management. For example, SALT allows comparison between versions of a class to be done using an external add-on.

SALT is included with a standard Dyalog APL installation for Windows but needs to be activated before it will be loaded when APL starts. See the section on **Configuration** below.

Implementation Details

File Format

Each version of each source object is stored in a Unicode text file with an extension of ".dyalog" by default. The Unicode file format used is known as UTF-8. These files can store text which uses the "Basic Multilingual Plane" of Unicode, which contains most of the world's languages and the APL character set. This format is supported by very many applications (including Windows Notepad).

The source code for SALT is itself "*salted*", consisting of two files in the SALT\SALT folder.

Application files are stored in text files with an extension of ".dyapp". When Dyalog APL is installed, it sets up associations for the new extensions:

.dyapp	Opens with Dyalog APL, which should Boot the application. Edits with Notepad.
.dyalog	Opens with Notepad.

Configuration

The registry key "HKEY_CURRENT_USER\Software\Dyalog\Dyalog APL/W 12.1\SALT" contains some values which provide configuration information for SALT. SALT will be loaded into the session if the registry string **SALT\AddSALT** has the value "1" (the default). If SALT is active, you should not get a VALUE error if you type

```
      ⎕SE.SALT
```

If SALT is not enabled you can enable it using the SALT workspace. Simply type

```
      )LOAD SALT
      enableSALT
```

With V12 you can also use the configuration menu to en/disable SALT but the effect is only effective at the next APL startup.

There are only a few settings and they can all be changed via SALT's *Settings* command shown later on.

Shaking the SALT

SALT is used to maintain itself, and the source can be found in the "SALTed objects" in the Dyalog folder SALT\SALT. As mentioned earlier, these files are loaded if the registry entry **SALT\AddSalt** has the value "1". If so, `SALTUtils` and `SALT` are loaded into `⎕SE`, and the function `SALTUtils.EditorFix` is connected to a callback on exit from the Dyalog Editor.

When SALT loads an object, it tags it with data. For class 9 objects it inserts a special namespace named SALT_Data in it, and variables inside this namespace contain the source file name, the version number and the last write time of the file when it was loaded. The last item of information is used to prevent accidental updates of the same version by two different users or from two different sessions. For class 3 & 4 objects a special comment is appended to the code with similar information.

SALT Applications

In addition to managing individual source code files, SALT is able to load and run applications defined by files with an extension of ".dyapp". The format of these files is documented under the Boot command. The Dyalog installation sets Dyalog APL as the application which is used to "start" these files, and SALT examines the command line

Comparing Files

SALT has its own built-in comparison program but since the source code for each version of an object is stored in a Unicode file, any file comparison tool which can compare Unicode files can be used. The author did a quick search on Google and ended up evaluating and subsequently spending $29 on a product called Compare It! from http://www.grigsoft.com/index.htm (if you download this product, make sure to get the Unicode-capable version). SALT is able to use any product if it can be launched using a command which takes the names of the two files to be compared as parameters. To inform SALT to use the above product you should do:

```
      ]SE.SALT.Settings 'compare [ProgramFiles]Compare It!\wincmp3'
```

SALT appends the two file names and calls "Compare It!" If such a program is unavailable then SALT will use it own primitive comparison code and show the results in the session.

Versions

SALT files may be *versioned*. When versioning is switched ON for an object, SALT creates files which have a version number immediately before the .dyalog extension (for example, MyClass.3.dyalog). The **List** function in SALT shows this number as [3].

Each time an object saved in a versioned file is changed, a new file is created. You can quickly end up with a large number of intermediate versions. You will need to use **RemoveVersions** to tidy up.

Using SALT

A standard Dyalog APL installation contains a collection of classes which can be used to explore object oriented programming. SALT commands allow you to explore and use this library.

List

The **List** command takes an object or folder name as its argument. An empty argument will list the top-level files and folders (immediately below the first folder named in the *workdir* setting[1]):

```
      ]SE.SALT.List ''
 Type   Name     Version  Size  Last Update
 <DIR>  Dyalog                 14-05-2006 21:32:54
 <DIR>  Samples                11-05-2006 08:21:15
```

If you get a VALUE ERROR when you try to use SALT functions, check that SALT is enabled (see **Configuration**, above).

The Dyalog folder contains official Classes Library supported by Dyalog. s*tudy* contains code which is referenced in documentation, or provided for self-study, *lib* contains SALT utilities, *tools* contains various developers tools. *spice* contains basic User Commands, more on that in the User Commands document.

The **List** function takes a number of modifiers. All SALT functions can be called with a single '?' argument, in which case they remind you of the available modifiers:

```
      ]SE.SALT.List '?'
```

[1] This setting may contain more than one folder but they must be separated by a semi-colon. When listing a file or folder the first one found is listed.

```
List pathname
Modifiers:
-Full[=1|2]     1 shows full pathnames below first folder found;
                2 returns "rooted" names.
-Recursive      Recurse through folders
-Versions       List versions
-Folders        Only list folders
-Raw            Return unformatted date and version numbers
```

We can get a complete list of class folders as follows:

```
      ]SE.SALT.List '-recursive -folders'
 lib
 SALT
 spice
 study
 study\data
 study\files
 study\GUI
 study\math
 study\OO
 study\OO\QuickIntro
 tools
 tools\DanB
 tools\SJT
```

List the contents of the study\OO\QuickIntro folder:

```
      ]SE.SALT.List 'study\OO\QuickIntro'
 Type  Name     Version  Size  Last Update
       Product           570   08-05-2006 08:59:45
       Sale              756   05-05-2006 13:15:20
```

Load

The **Load** command takes an object name or a pattern as its argument:

```
      ]←]SE.SALT.Load 'study\files\ComponentFile'
#.ComponentFile
      cf←]new ComponentFile 'c:\temp\compfile'
      cf.Count
2
      ]←]SE.SALT.Load '\myutils\gui*'
guiin guimsg …    guiout
```

Load returns a shy reference to the loaded class(es) or a message for functions. By default, **Load** also gives the loaded class/namespace a "global name", in this case ComponentFile, where it has been called. See the description of the **New** command below for a description of the **–noname** option, which allows you to avoid the creation of the global name and use a class "without loading it into the workspace".

Load's modifiers are:

```
      +]SE.SALT.Load '?'
```

```
Load [path]classname
Load a class in the workspace

Modifiers:
-Target=Namespace      Specifies target namespace for load
-Disperse[=nam1,nam2]  Disperse elements in the Target ns specified
-NoName                Only return ref, do not create name
-NoLink                Do not "link" loaded class to source file
-Version=              Load specific version
-Source                Return the source
```

The**–Target** modifier allows you to load a class into a particular namespace:

```
      'MyFiles' ⎕NS ''
      ⎕←⎕SE.SALT.Load 'study\files\ComponentFile -Target=MyFiles'
#.MyFiles.ComponentFile
```

The -**Disperse** modifier allows you to bring in the objects that are IN the file as opposed to the object itself into the target namespace. If only specific objects need to be brought in they can be specified after as in –disperse=obj1,obj2,etc. The result of **Load** in this case is a message explaining that it did or why it did not do it. No tracking information is kept in this case (see **NoLink** below).

```
      ⎕←⎕SE.SALT.Load'GUIutils -disperse'
15 objects dispersed
```

The **–Version=** modifier allows you to load a particular version of an object, we'll show examples of this a bit later.

The -**Source** modifier will make Load return the source instead.

Finally, you can use the modifier **–NoLink** to specify that SALT should not insert tracking information into the object. If you use –**NoLink**, editing a SALTed object will NOT cause SALT to offer to save the source upon exit from the editor. If you wish to save the object again you will need to use the **Save** or the **Snap** command.

New

The **Load** command takes a modifier called **–NoName** which allows you to specify that you do not want the global name created. This allows you to use a class without giving it a name in the workspace. The following example defines an unnamed class which is used to open a component file and return the number of components, but leaves no trace in the active workspace:

```
      ⎕EX 'ComponentFile'
      (⎕NEW (⎕SE.SALT.Load 'study\files\ComponentFile -NoName')
                                          'c:\temp\compfile').Count
2
      ⎕NC 'ComponentFile'
0
```

The **New** command provides a more direct way to instantiate objects from a source file:

```
      cf←⎕SE.SALT.New 'study\files\ComponentFile' 'c:\temp\compfile'
      cf[1]
```

```
comp 1
```

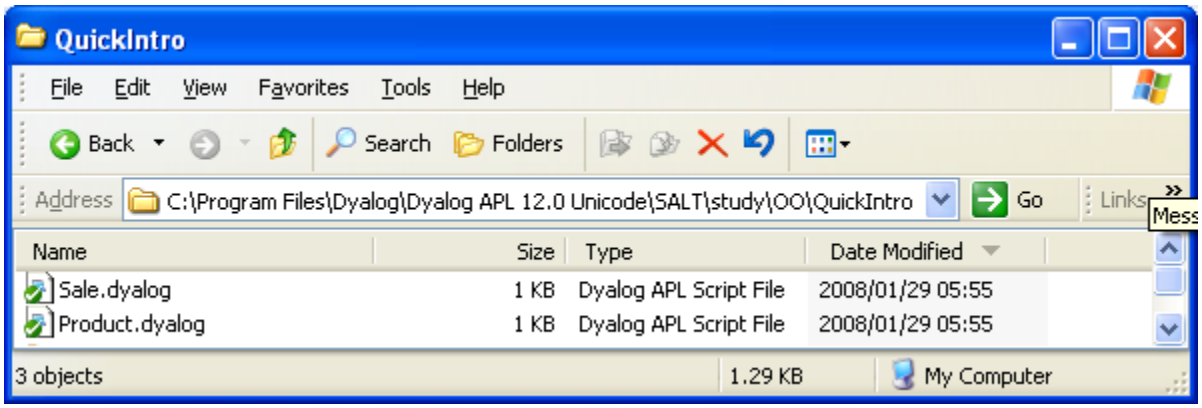
New passes the first element of its argument to Load, appending the **–noname** option, and then makes a new instance of the loaded class using the rest of the argument.

Explore

As an example of using SALT, we are going to load one of the QuickStart example classes, modify it and save it under a new name. The **Explore** command can be used to open Windows Explorer up on a folder, for example:

```
      ]SE.SALT.Explore 'study\OO\QuickIntro'
```

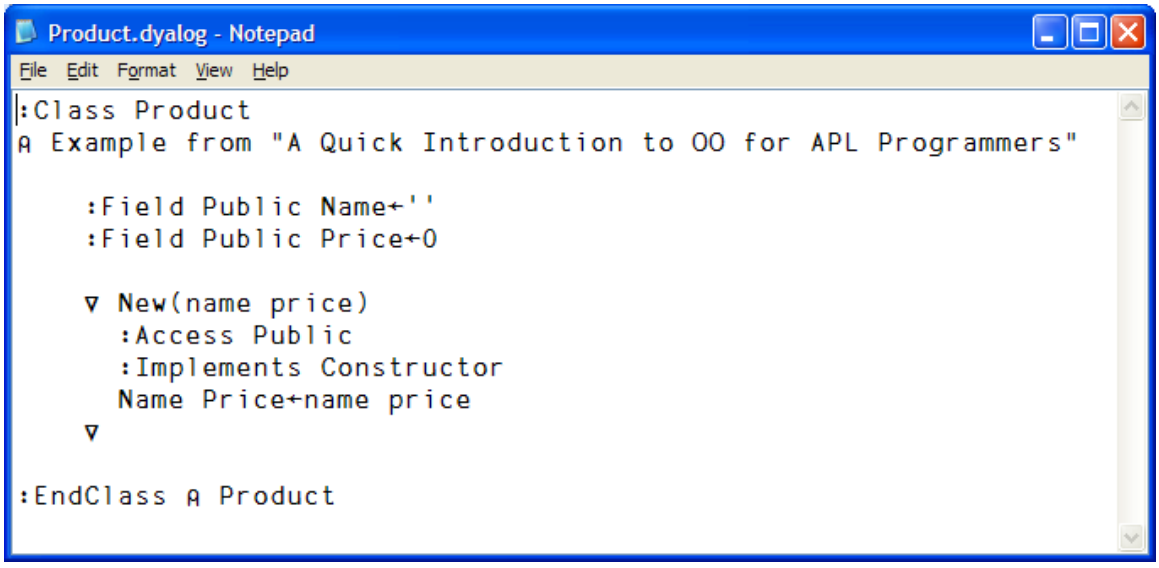
You should see explorer open up to show the contents of the Dyalog folder:



If the argument to the Explore command is an object name rather than a folder name, Explore will start the Windows Notepad:

```
      ]SE.SALT.Explore 'study\OO\QuickIntro\Product'
```

This is the class we are going to experiment with:



Load it into the workspace and edit the class, changing its name to *MyProd*:

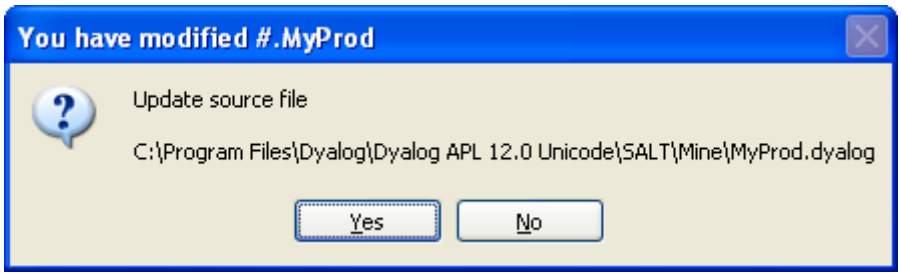
```
      ⎕←⎕SE.SALT.Load 'study\OO\QuickIntro\Product   -nolink'[2]
#.Product
      )ed Product
```

Save

We are now ready to save our class called *MyProd* in the folder Mine (which does not exist yet, so we use the –**makedir** modifier). Save returns the full name of the file which was created:

```
      ⎕SE.SALT.Save 'MyProd Mine\MyProd  -makedir'
C:\Program Files\Dyalog\Dyalog APL 12.0\SALT\Mine\MyProd.dyalog
```

Now, edit MyProd again and make a small change – for example to the comment. As you exit from the editor, you should see a pop-up similar to the following:



Click **Yes** and use Notepad and Explorer to verify that the file contains the new version of MyProd[3]. Experiment with clearing the workspace, loading Mine\MyProd, and verifying that all the changes you make are being saved in the file.

Save accepts other modifiers:

```
      ⎕se.SALT.Save '?'
Save class filename
Save class in a specific file

Modifiers:
-Convert        Convert the namespace into source form if necessary
-Noprompt       Do not prompt for confirmation
-MakeDir        Create any necessary directories
-Version[=]     Version number to save
```

Normally non scripted namespaces cannot be saved but SALT manages to do it by creating a temporary script that can be put onto file[4].
With –**convert** you get to keep the new format and, as a bonus, SALT can keep track of the changes and save automatically when you edit the script.

-**noprompt** allows you to skip the confirmation window and save without it

---

[2] we add the –NOLINK modifier because we don't want SALT to track the changes
[3] There is a way to prevent SALT from asking confirmation each time you edit a script, see **Settings** below
[4] there are restrictions and embedded GUI objects, for example, will prevent a successful save

Versions

By default, SALT maps your class or namespace to a single file, and any change you make to the object overwrites the file. If you give your object a version number, SALT will start taking backup copies each time you make a change (note that modifiers can be abbreviated, as long as they are uniquely identified):

```
      ⎕SE.SALT.Save 'MyProd -ver=1'
C:\Program Files\Dyalog\...\SALT\Mine\MyProd.1.dyalog
```

When SALT notices that you are giving an object a version number for the first time, it starts saving the existing class under a name that includes a version number.

Each time you change MyProd and update the file, you will see a message confirming the creation of a new file with a version number up one from the previous version. Make one or two more changes to MyProd and then call List with the -**Versions** modifier :

```
      ⎕SE.SALT.List 'Mine -vers'
Type  Name    Version  Size  Last Update
      MyProd  [2]       301  2008/02/02  9:28:30
      MyProd  [1]       301  2008/02/02  9:27:05
      MyProd            301  2008/02/02  9:26:08
```

Imagine that we are now planning a new release of MyProd, which we are going to save under version 10, skipping a few versions. We start the version 10 project by saving the new version:

```
      ⎕SE.SALT.Save 'MyProd -version=10'
C:\Program Files\Dyalog\…\SALT\Mine\MyProd.10.dyalog
      ⎕SE.SALT.List 'Mine'
 Type  Name    Version  Size  Last Update
       MyProd            301  2008/02/02  9:29:32
```

Since we haven't made any changes to MyProd yet, this version is identical to the last one. To see all versions we need to include the –**version** switch:

```
      ⎕SE.SALT.List 'Mine -ver'
Type  Name    Version  Size  Last Update
      MyProd  [10]      301  2008/02/02  9:29:32
      MyProd  [2]       301  2008/02/02  9:28:30
      MyProd  [1]       301  2008/02/02  9:27:05
      MyProd            301  2008/02/02  9:26:08
```

Now, change the constructor function in MyProd so that it sets the display form for the instance (this will create version 11), for example:

```
    ∇ New(name price)
      :Access Public
      :Implements Constructor
      Name Price←name price
      ⎕DF'[',(⍕Name,'@',(⍕Price),']'
    ∇
```

By default, the **Load** command will load the most recent version of an object. Verify that Load is loading the latest version by default, but that the other versions are still available:

```
      ⍞←⎕SE.SALT.Load 'Mine\MyProd'
#.MyProd
      ⎕NEW MyProd ('Widget' 100)
[Widget@100]
      ⎕SE.SALT.New 'Mine\MyProd -ver=1'  ('Widget' 100)
#.[MyProd]
```

If we combine the -**version** and -**noname** modifiers, we can in fact work with multiple versions of the same class at once.

```
      pclasses←{⎕SE.SALT.Load 'Mine\MyProd -noname -ver=',⍕⍵}¨1 11
      pclasses
 #.MyProd  #.MyProd
      pclasses.SALT_Data.Version ⍝ SALT version tags
 1  11
      {⎕NEW ⍵ ('Widgets' 100)}¨pclasses
 #.[MyProd]  [Widgets@100]
```

Snap

The **Snap** command allows you to do bulk save. It supports some of Save's modifiers (**makedir**, **version**, **noprompt**) plus others to save selectively. It even can produce a .dyapp script or function to reload all the workspace. For example:

```
      )LOAD myutils
      ⎕SE.SALT.Snap '\store\here -makedir -loadfn -vars '
```

will save everything in the workspace[5] in individual files under <*store**here*> folder and produce a function to reload the workspace like this:

```
      ⎕SE.SALT.Boot '\store\here\load_ws.dyalog'
```

See the appendix on **Snap** at the end of this document for full details.

Compare

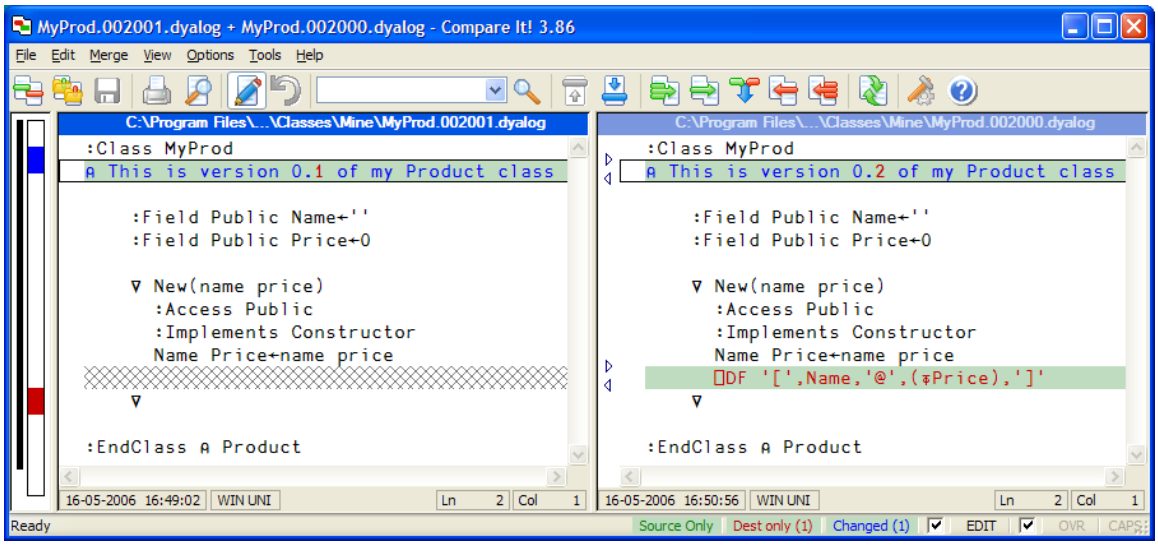
The Compare command allows you to compare versions of an object. Note that if you want the Compare command to use a third party product like the Unicode version of "Compare It!" or a different file comparison tool, you must use the **Settings** command or, in V12, use the configuration menu to specify where it is. If you leave this setting empty APL will use its own simple comparison function.

By default, Compare shows you the differences between the 2 most recent (highest) versions of the file given as argument (here *MyProd*).

```
      ⎕SE.SALT.Compare'Mine\MyProd'
```

Should bring up a screen which looks like this if you are using "Compare It!":

[5] There are restrictions, for example GUI objects cannot be saved



Compare also takes a modifier which allows you to specify exactly which versions you want to compare. You can compare the code for version 3 to the most recent version using:

```
      ⎕SE.SALT.Compare 'Mine\MyProd -ver=3'
```

If you want to compare two non recent versions, you need to provide 2 version numbers, for example:

```
      ⎕SE.SALT.Compare 'Mine\MyProd -ver=1 10'
```

If you want to compare the latest version of a class with a class with the same name IN THE WORKSPACE you can specify –version=ws:

```
      ⎕SE.SALT.Compare 'Mine\MyClass -version=ws'
```

Using a different program

Should you decide to use a program other than the one specified in the registry to perform the comparison you can use the -**use** switch to specify which program to use. For example, if you have 'Beyond Compare' (another comparison tool from the Net) installed and you just want to try it you can do

```
      ⎕SE.SALT.Compare 'Mine\MyProd -use=[ProgramFiles]\BC\BC2.exe'
```

This will not change your registry entry and subsequent use of Compare will use whatever setting you currently have set in your session.

RemoveVersions

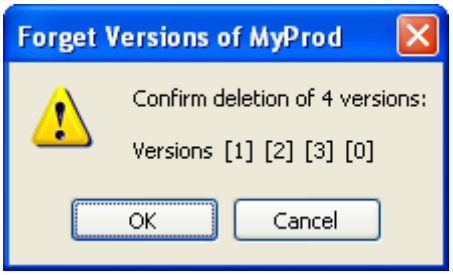
SALT creates a new file every time you edit a class or namespace. Therefore, you need to clean up versions occasionally.
This command takes three modifiers, two of which are mutually exclusive:

-**Version**=n Specifies the version(s) which should be deleted
-**All** All but the last version should be deleted

For example (version 0 is the original version):

```
      ⎕SE.SALT.RemoveVersions 'Mine\MyProd -ver= <4'
```

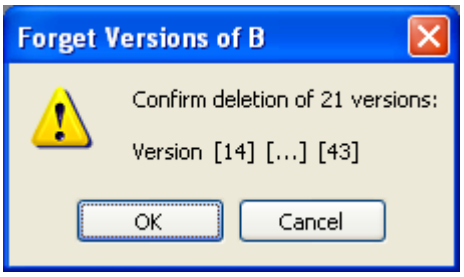
produces




```
4 versions deleted.
```

You can also delete trailing versions. If trailing versions are deleted they can be collapsed into one using –**collapse**. For example, suppose you have been working of a script starting at version 13 and you are happy with the result after you have made many modifications, only the last of which (version 43) you want to keep. Instead of listing all the versions to remove as in –ver=14 15 16 17 18 19… you can type

```
      SE.SALT.RemoveVersions 'Mine\MyProd -ver=>13 -collapse'
```




```
20 versions deleted.
1 version renamed
```

Versions 14 to 42 are deleted and version 43 becomes version 14.
In any case (-**all** or -**ver**=) versioning resumes at the highest number+1 when changes are made.

If the number of versions is too long to be described, e.g. from 3 to 64 you can use a dash. For example, to delete versions 7 to 99 you use

```
      SE.SALT.RemoveVersions 'Mine\MyProd -ver= 7-99'
```

Boot

With the **Boot** command, you can use a script file to describe the loading and initialization of an application – as an alternative to using a saved workspace. The Boot command reads files with the extension *.dyapp*. or *.dyalog* Each line of a .dyapp script is either a SALT **Load** command or a **Run** command. There should be at least ONE line with the word *Run*, followed by the name of a method to call.

For example, a .dyapp file might read as follows:

```
Load study\files\ComponentFile
```

```
Load study\files\KeyedFile
Load MyApp
Run MyApp.Main
```

All scripts are loaded into #.
Boot can also run a function in a *.dyalog* file. In that case the function is run 'as is' but will be given the 2nd element of the argument if it takes an argument. The result will be discarded. For example, assuming monadic function <X> is in file /sale/x.dyalog:

```
      ⎕SE.SALT.Boot '/sale/x -function' 'ABC'
```
or
```
      ⎕SE.SALT.Boot '/sale/x.dyalog' 'ABC'
```

will run localized <X> with argument 'ABC'. If the function does not take an argument then 'ABC' will be ignored.
This will often be used in conjunction with **Snap** which can produce functions to be used for **Boot**. In that case the code will include a statement to execute []LX. To avoid this add the switch **–xload**. See **Snap** at the end of this document for details.

Note that, if there are dependencies between classes (as above, where KeyedFile derives from ComponentFile), base classes must be loaded before any classes which derive from them. SALT does not perform any dependency analysis but you can include statements to tell SALT to load other classes before. For example, if script A requires script B you should add this statement somewhere in A:

```
⍝∇:require path\B
```

Where **path** may be replaced by '=' to signify '*same path as mine'*.

Autostarting SALT Applications

If SALT is active, and APL is started with the name of a .dyapp file on the command line instead of an APL workspace, SALT initialization will call the Boot command on the named file. In this way, a .dyapp file can be used to auto-start APL applications which are based on SALT. Note that the whole application does not need to be "salted": Once started, the application can use ⎕CY or other mechanisms to bring in additional source code.

Platform independence

SALT should perform the same way under Windows® and *nix platforms. To avoid confusion for people dealing with both environments SALT will accept SALT pathnames (only) using either / or \ as folder separator.

Under Unix there exist a version without GUI, which works in "terminal" mode. Under that system SALT must be enabled manually through the workspace 'salt'. Simply)LOAD salt and use <enableSALT>.
The workspace can also be)LOADed at startup time, just like any other workspace, by issuing the apl startup command followed by the path of the salt workspace as in

```
startapl ws/salt
```

If another workspace must be)LOADed afterwards or if a .dyapp file must run after simply put it in between, e.g.:

```
startapl  anotherws  ws/salt
```

Settings

Some commands require *global* parameters. For example, the **Compare** command needs to know which program to run to perform the comparison. This information is taken from the registry and loaded into SALT at boot time. It becomes a *session* parameter and can be modified using the **Settings** command.
In some cases Settings can also be specified on the line with the command using the –**USE**= switch only for the duration of the command.
For example if the default **Explore** program is not satisfactory and you want to try another one, say, vi.exe, then you can specify it on the command with –use=\myprogs\vi.exe
If you find this is useful you may want to make the setting for the duration of the session by entering

```
      ]SE.SALT.Settings 'editor \myprogs\vi.exe'
```

Should this prove unacceptable you can enter

```
      ]SE.SALT.Settings 'editor -reset'
```

to reload the value from the registry. On the other hand if those values are quite acceptable and you wish to make them permanent you can issue

```
      ]SE.SALT.Settings 'editor -permanent'
```

and the registry will be altered accordingly.

To see the list of all settings enter

```
      ]SE.SALT.Settings ''
```

Other settings are **workdir** and **edprompt**.

workdir allows you to have multiple working directories separated by semi-colon. To add a directory use comma, to remove one use ~, like this:

```
      ]SE.SALT.Settings 'workdir ,\proj\p1' ⍝ add \proj\p1
```

from then on files are stored under \proj\p1 but retrieved from where they are first found in the list of directories. SALT's files are always assumed in **[Dyalog]\SALT** even if that path has been removed.

edprompt determines whether you are prompted for confirmation to overwrite the file each time you make a modification to a script. The default of 1 prompts you each time.

Conclusion

The Simple APL Library Toolkit (SALT) provides basic source code management features for APL classes and namespaces stored in Unicode script files. By themselves, Classes provide new ways to make code sharing easier within the APL community. However, we believe that the full benefit of Classes will only be felt by the community if it also has a common source code management system, or at least a common file format which can be manipulated by a family of tools.

For all its complex definition SALT is deceptively simple; it merely SAVES and LOADs with a few utilities.

We hope that SALT will prove to be powerful enough that many users of Dyalog APL will decide to use it in real applications – at least as a tool to load shared utilities - and that it can be the beginning of a simple common source code management system which will provide the required platform for APL users to share utility classes and namespaces more effectively than they have been able to do so in the past.

Appendix A - The Snap command

Syntax:
Snap [path] –class= –convert –fileprefix= –loadfn[=] –makedir –noprompt –pattern= –show[=] –vars[=] –version

In the following text *object* refers to any APL object, either code (function, op) or data (including all namespaces).
Snap by default saves all new objects in the current namespace to the path specified as argument and saves modified objects to their proper location. It returns the list of the names of the objects that have been saved, if any. If the path is not specified then the current *workdir* is used.
Snap, in its present state, cannot save everything and variables in #, function refs and instances cannot be saved.
Snap uses <**Save**> to store objects.

Procedure

Snap first finds the list of all objects and from those the ones that can be saved. It then determines which ones have been modified or are new. If any of them needs saving or if it cannot determine if they need to be saved (e.g. non scripted namespaces) then each one of these object is saved using <**Save**>.

To find out which object needs to be saved SALT marks objects (functions and namespaces) with special tags when it loads them or when they are first saved. Because # variables cannot be tagged they cannot be saved in canonical form on file. Variables in namespaces, on the other hand, can be saved as part of the namespace without any problem as long as they don't contain anything 'illegal' like forms, []ORs, etc.

Snap can save non-scripted namespaces by making them into a script beforehand, but tagging information cannot be retained and **Snap** will always overwrite (given permission) the existing file where they reside. **Snap** is non destructive in that respect and unless told otherwise (see below) it will keep the original non-scripted form even though it writes a source for them.

Alternate behaviour

Snap accepts a series of modifiers to alter its operation.

Selecting the objects to save

It may be desirable to save only a subset of the workspace, for example only the functions or names starting with a specific pattern.

To select specific name classes (e.g. functions or D-ops) you use the -class= modifier. It takes a value of one or more classes. Acceptable classes are 3=all functions, 4=all ops and 9=all namespaces. Finer numbers are also accepted, e.g. 4.1 for trad ops or 9.5 for interfaces.
To select objects following a specific patter you use the –pattern= modifier. It takes a string where '*' means "any number of chars". At the moment the pattern matching is restricted to strings ending in '*' and stars are ignored, effectively meaning "only names starting with the string given before the star" as in 'util*'[6].
Both can be combined and the following with save all namespaces starting with 'GUI':

```
      []SE.SALT.Snap '/ws/utils  -class=9.1  -pattern=GUI*'
```

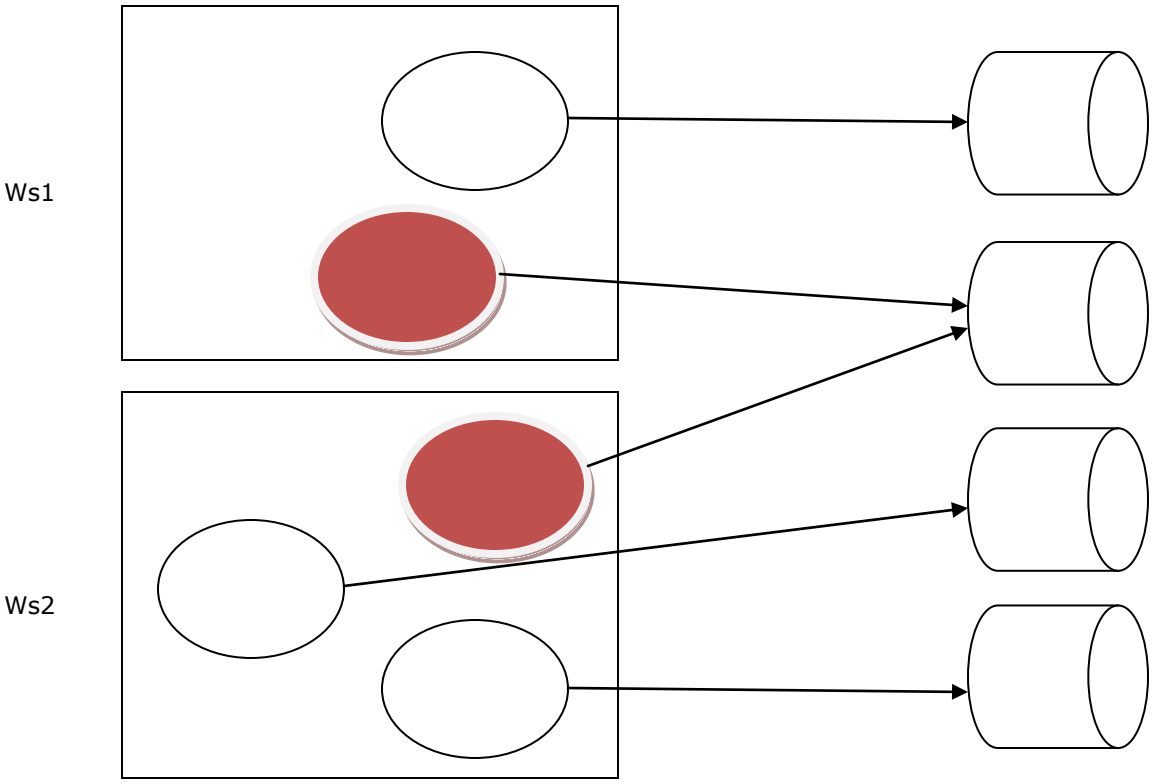
[6] Without the star the pattern would be an exact match for the name specified in which case the <Save> command might as well be used

Both also accept the character '~' in the first position to **exclude** specified class or pattern, e.g.

```
      []SE.SALT.Snap '/ws/utils   -pattern=~GUI*'
```

will save all objects NOT starting with 'GUI'.
These modifiers allow you to save various subsets of functions/namespaces to different locations. Two or more different workspaces can use the same subset which can be updated independently of the workspaces.

How two workspaces can share code



Verifying what will happen.

If you are not sure what will happen when you use **Snap** you can ask it to show you which objects would be saved only. For example,

```
      []SE.SALT.Snap '/ws/utils  -class=~9  -show'
```

will only show you the names that would be saved without actually saving them. If you also want to know where they would be saved you can use '–show=details'.

Changing the filenames

By default the filenames used are the same as each object's name followed by '.dyalog'. If you wish to prefix these names by a special string you can use the –fileprefix modifiers. For example,

```
      []SE.SALT.Snap '/ws/utils   -pattern=GUI* -fileprefix=Win'
```

will save all new objects starting with 'GUI' to files starting with 'Win', thus function 'GUImenu' will be saved in file '/ws/utils/WinGUImenu.dyalog'.

Skipping prompts

Unless your general prompting settings are set to NO, **Snap** will prompt you for replacement each time it finds a file that already exist. You will then have a choice of **Yes** to replace the

file, **No** to skip that file or **Cancel** to skip the rest of the files. **Snap** will return only the names of the objects that have been effectively saved. If you wish to skip the prompting you can add the modifier –**noprompt**.

Using version numbers

SALT has the ability to keep version numbers for each file. If you wish to enable that feature for the objects **Snap** is about to save use the modifier –**version**. You can also specify the version number to use in which case ALL objects will be saved with the same version number.

Creating intermediate directories

When **Snap** saves an object it assumes the directories in the path given as argument already exist and won't attempt to create them. If you know they don't exist (typically the first time you save objects) you must ask **Snap** to create the directories for you by using the –**makedir** modifier otherwise the command will fail.

Converting namespaces to scripted form

When **Snap** saves namespaces in non-scripted form it has to convert them first into scripted form in order to perform the *save*. It then gets rid of the converted format and keeps the original one. If you subsequently **Snap** the workspace it will redo the work even is no change occurred to these namespaces because they cannot be tagged to let know **Snap** of the changes. If you wish to retain that form use the –**convert** modifier.

Recreating the workspace

You may be interested in recreating the workspace at a later moment in time. To do so you must remember which objects made up the workspace and bring them back one by one. Doing this by hand would be too tedious and prone to errors so **Snap** provides a way to do that for you.

If you use the –**loadfn** modifier **Snap** will create a function named <load_ws> in the same path given as argument. That function, when executed, will bring every object needed in the current workspace[7] and run the []LX. If you want the function to exist elsewhere use –**loadfn**=*/new/path* instead. If the name 'load_ws' does not suit you then you can change it by adding an extension. There are 2 possible extensions: '.dyalog' (or more simply '.') which will create a function or '.dyapp' which will create a script. Both are suitable for use by **Boot** and have the same effect. For example, to create a function named <loadit> do

```
      []SE.SALT.Snap '/ws/myapp   -loadfn=/ws/myapp/loadit.dyalog'
```

Trick

If the location of the function to reload the workspace is the same as the code of the workspace you can avoid re-entering the same pathname for the *loadfn* modifier by replacing the path by '='. And use only '.' instead of '.dyalog'. Using the example above it would become

```
      []SE.SALT.Snap '/ws/myapp   -loadfn= =/loadit.'
```

You can bring that (load) function in a workspace to execute it or ask SALT to do it for you via the **Boot** command. **Boot** takes a path to a file to read in and run. If the path is a function you should add the '.dyalog' extension or use the –**function** modifier to remove any ambiguity. For example:

```
      []SE.SALT.Snap '/ws/myapp   -loadfn  -makedir' ⍝ store the ws
      )CLEAR
      []SE.SALT.Boot '/ws/myapp/load_ws -function'   ⍝ bring it back
```

will save the workspace (and create /ws/myapp if it did not exist) and recreate it in a clear workspace. The WSID and the []LX will be the same. []LX will be run automatically. If it should not be run then use the –**Xload** modifier to Boot.

[7] It will merge the objects with the existing ones in the workspace, replacing them if necessary

Trick

If you have several workspaces to snap whose directory name is the same as the workspace's name you can use '=' instead of the workspace name when specifying the target folder. For example, let's say you want to store the workspaces in your personal library under /apl/wslib so that workspace 'xyz' is stored under **/apl/wslib/xyz** you can use

```
)Xload xyz
[]SE.SALT.Snap '/apl/wslib/=   -loadfn  -makedir'
```

for each workspace without having to retype the **Snap** line for each workspace; simply grab the line from the session log to re-execute it after each)XLOAD.

Restrictions

In its present form **Snap** cannot store some objects like instances (e.g. GUI) or variables in #. This restriction is unlikely to be lifted soon. Function refs are another kind of object impossible to store in this version.

Workarounds

Recreating the variables or instances upon)LOADing the workspace (i.e. thru []LX) will always work but may not always be practical because sometimes the work involved in recreating them will be substantial. For variables another way to do this would be to store them in a separate (scripted) namespace. When the workspace is reloaded the variables are moved to the root space. There is a modifier to do just that.

This modifier, **vars**, will create a namespace with the definitions of all legal root variables and save it under the name 'vars_ns'. When used in conjunction with the **loadfn** modifier the load function created will contain a statement to load the contents of the file and disperse its contents into root, effectively recreating all globals. If you prefer another name for the file you can use –**vars**=newname

Epilogue

For all its elaborate definition Snap is fairly simple. It merely saves all new objects where specified and modified ones where they belong. All the modifiers simply allow you to fine tune the process.

Dyalog 2009