Documentation - A Must

Overview

Documentation is often coming on paper. Only problem is that paper cannot be debugged: mistakes are less likely to be encountered in documentation than in code.

Under pressure, when an application is enhanced or debugged, the documentation often is not updated. I found a faulty, out-of-date documentation worse than having no documentation at all. Let's look into this in detail.

Generally, there are two quite different kinds of documentation:

- The "big picture", which tells the why and how, and maybe the strategy a solution is based on.
- A more technically oriented documentation, in APL often about function calls, parameters and function call results.

The great picture needs to be written on paper or a web site or the like. By the way, you are just reading a kind of big-picture-documentation: this paper tries to put you into the picture about ADOC, the class "The Open Source APL Project" (TOSAP) was started with. This paper is focussing on the technical oriented documentation.

Classes: the Public Interface

It is possible to gain useful information from a class definition even if there is not a single comment in the script. This is possible because classes offer a kind of public interface: methods you can call and fields and properties you can reference or even set if they are not defined as read-only.

Apart from this public interface there usually is much more in a class definition, but it is hidden for good reasons. OO is believed to offer serious advantages. One is that the user of a class does not need to know or care about the implementation details.

That is the reason why investigating a class definition written by somebody else should normally not happen by looking into the class definition, or using the tracer, because that will not only display the public interface but also all the internal stuff.

What is needed is a mechanism to extract the public information only, ignoring anything else.

By extracting all kinds of useful information from a class definition, questions like the following can be answered:

- Which fields can be read and set, what defaults are used, and which are read-only?
- Which properties can be read and set, and which are read-only?
- Which methods, properties and fields are of type "Instance", and which are of type "Shared"?
- Which methods are available, and which syntax do they have?
- Which namespaces are included?
- Which namespaces are requested?
- Which constructors are available, and which syntax do they have?

- Is there a destructor?
- From which class is the class in question inheriting?

That's exactly what ADOC is doing, but ADOC can do even more if you add appropriate information to a script.

Adding comments

If the class author obeys two simple rules, even more useful pieces of information can be extracted automatically from a script:

- What is a particular class or interface or scripted namespace good for?
- What is a particular field good for?
- What is a particular property good for?
- What is a particular method designed to do, and which parameters will it take/accept and how will the result will look?

Whether it is a good idea to add lots of information to a script or not depends on several parameters like size and complexity of the class as well as your personal view. Anyway, let's discuss the rules in detail.

The class description

Any subsequent lines after the ":Class" line where the first non-blank character is a lamp are taken as a general description of the class, while empty lines as well as lines starting with either ":Implements" or ":Access" are ignored.

An Example:

```
:Class Myclass
⍝ These comment lines should give you an idea what
⍝ this class is supposed to do.
:Access Public Instance
⍝ Author: Mr. Foo

⍝ more comments
⎕ML←3
⍝ even more comments
…
```

While the empty line and the ":Access" line are both ignored, the first 4 comment lines are taken as the general description of the class. According to the rule, the line following the setting of []ML is ignored.

Later we will discuss how you can format the text according to your needs.

Properties

The rules for properties are very similar to the one we have just discussed. An example:

```
:Property MyProperty
⍝ line 1
⍝ line 2
∇ r←get
r←global
∇
⍝ line 3
```

The comment lines 1 and 2 are taken as a general description of "MyProperty", but the third line is ignored: only subsequent comment lines are taken into account.

Methods

Methods are offering a kind of service useful to manipulate data inside the class itself (if shared) or any instance of a class, or extract data from either the class or any instance. Because the syntax of a method is important, it is reported as well. Furthermore, all subsequent lines following the method header where the first non-blank character is a blank are taken as a general description of that method. Again, all blank lines as well as lines starting with either ":Implements" or ":Access" are ignored.

```
r←MyMethod y
⍝ Line 1
⍝ Line 2
:Access Public
⍝ Line 3
```

All three comment lines are taken as a general method description.

Fields

Fields are handled in a slightly different way. Comments are only taken from the definition line:

```
:field Public shared MyField←1 2 3 ⍝ a comment about "MyField"
⍝ to be ignored
```

Because fields can be initialized in the ":Field" line, any settings are reported, too.

Formatting

You can use HTML to make the formatting fit your needs. See the following example:

```
:Class MyMethod
⍝ <2>A Caption
⍝ These comment lines should give you an <b>idea</b> what _
⍝ this class is supposed to do.
⍝ Author: Mr. Foo
⍝ <h3>A sub-caption
⍝ <pre>(+/+/matrix)÷2⍴⍴matrix</pre>
⍝ An ordered list:
⍝ * line 1
⍝ * line 2
⍝ An unordered list:
⍝ # any line
⍝ # another line
…
```

Some remarks:

- Note that <h{n}> tags do not need to be closed with </H{N}>
- The <h1> level is taken for the class name as the top header and should therefore not be used.
- By default, every line is converted into a separate paragraph. If you want to concatenate two lines, add ` _` (a blank followed by an underscore) at the end of the line which you want to be connected with the next one.

- Ordered lists as well as unordered lists can be inserted easily by letting a line start with '* ' (ordered) or '# ' (unordered). That's easier than putting lots of HTML text into the text.

- *Any* HTML code can be included. Note that code within <pre> tags is displayed with the "APL385 Unicode" font in a resulting HTML page if you use the default style sheets ADOC is coming with.

- These remarks hold true not only for the class description but also for the descriptions of fields, properties and methods.

What ADOC delivers

ADOC can be used to either generate a short kind of overview in the workspace, limited to the most important information, or an all-singing-all-dancing HTML page. Of course ADOC can generate a single HTML page for as many class definitions as you like in a single call. If more than one script name is passed as right argument, a table-of-contents is added to the top of the HTML page. This allows a user to jump quickly to the desired class documentation.

Short Overview

To get an overview about a particular class, execute this in the session:

```
      ADOC.List  ADOC
*** ADOC ***
Constructors:
  make_1 fullDocName
  make_2(fullDocName caption)
  make_0
Instance Properties:
  htmlFinalised (read only)
  BrowserControl
  Creator
  FullDocName
  HTML
  Meta (read only)
  WithRunningNumbers
Instance Fields:
  nl←⎕av[4 3]
  Caption←''
  CssPrint←'Print'
  CssScreen←'screen'
  MaxWidthInChars←40
Instance Methods:
  Analyze
  CreateDocFooter
  CreateDocHeader
  CreateHtml
  FinaliseHtml
  Reset
  SaveHtml2File
Shared Methods:
  Browse
  List
  ProcessAsHtml
```

As you can see, by default only the names of the methods are reported. If you are in need for the calling syntax of the methods you can use the "full" option:

```
'full' ADOC.List  ADOC
```

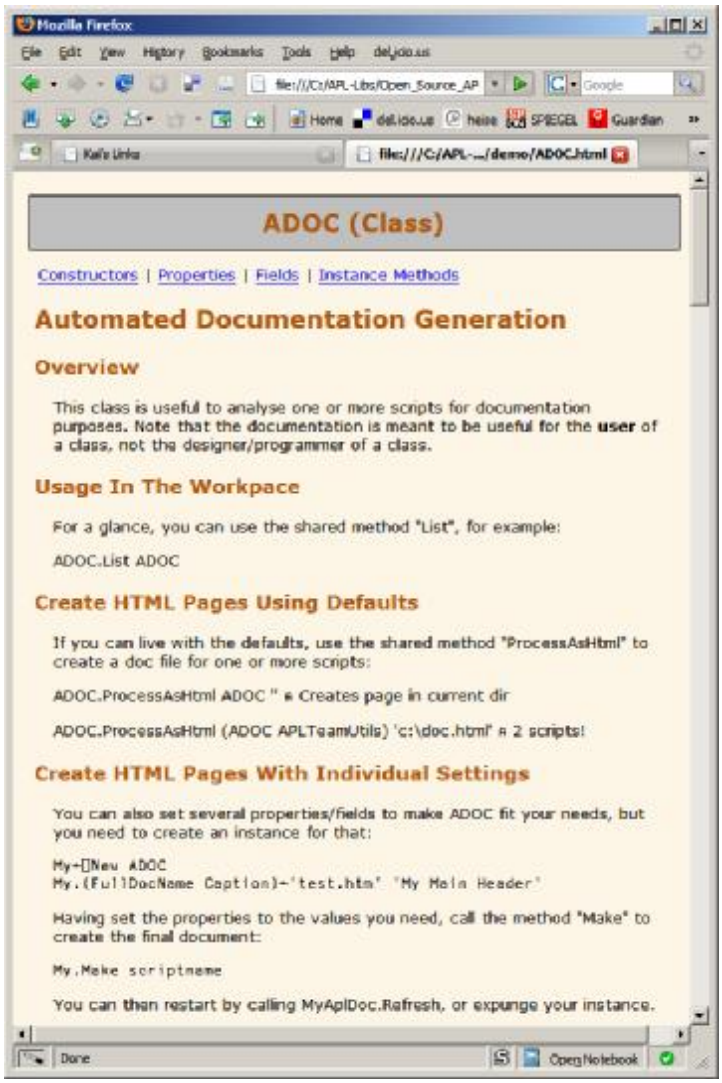
There are several ways available to create a full report.

Browsing a Class

To display a class on the fly, you can use the "Browse" method. For that you do not even need an instance:

```
ADOC.Browse ADOC
```

That will lead to this new window:



By default ADOC uses your default browser. If you prefer to use a different browser, you can do so by specifying the "BrowserName" property. Note that this property accepts only one of this: "IE", "IEXPLORE", "Firefox", "Opera", "Mozilla", "Safari".

You can use other browsers as well but you must specify the path to the exe with the "BrowserPath" property to achieve this.

The setting of "BrowserName" as well as "BrowserPath" is saved in the Windows Registry, so you need to do this only once per user/machine.

Parameters

ADOC.Browse uses sensible defaults, but of course you can change these defaults to whatever suits your needs. One way to achieve this is to create an instance, set the properties appropriately and finally call the appropriate methods. However, there is another way available to achieve this by using the shared method "Browse". You can create a parameter namespace and pass it as a left argument to "Browse":

```
parms←⎕ns ''
parms.Caption←'My Caption'
parms.Browser←'Safari'
parms #.ADOC.Browse #.ADOC
```

Using this technique you can specify the following properties:

- BrowserName
- BrowserPath
- Caption
- IgnorePattern
- Inherit
- embeddedClassName

Note that these parameters are ***case sensitive***.

Persistent Pages

If you want create a persistent version of the HTML page, you can use the #.ADOC.ProcessAsHtml method. This is a shared method, too. The method requests two parameters:

- The name of one or more scripts to be processed.
- The name of the file the HTML code is finally saved in.

#.ADOC.ProcessAsHtml comes with reasonable defaults. However, you can overwrite some of them (listed above under #.ADOC.Browse) using a parameter space just as you can with #.ADOC.Browse.

If this is still not good enough to match your needs you are supposed to created an instance of ADOC and to deal with everything on your own.

An example:

```
myDoc←⎕NEW #.ADOC (,⊂filename)
myDoc.(Caption FullDocName)←'foo' 'test.htm'  ⍝ Set parameters
myDoc.Make (class1, class2)
```

You can then either expunge the instance or call the myDoc.Reset method and process other script(s).

You can specify a parameter space as the left argument of "ProcessAsHtml" as you can for "Browse" – see there. However, note that the property " embeddedClassName" is not available here.

Parameters and Fields

The easiest way to get up-to-date information about this is to process the ADOC script with ADOC – nothing else to say.

The Style Sheets (CSS)

ADOC uses CSS (Cascading Style Sheets) to beautify the resulting HTML page. By default, ADOC creates such style sheets dynamically and embeds them into the HTML code. You can specify your own style sheets by specifying "screen.css" and/or "print.css" and saving them in the same directory the ADOC script is loaded from.

Prerequisites

ADOC needs the class "UnicodeFile" Dyalog comes with. ADOC tries to find this class in []SE. If the class cannot be found there, it tries to find it in #. If it is not found there either, a VALUE ERROR is reported.

Installation

To enable ADOC to find its style sheets, copy the script "ADOC.dyalog" files into %DYALOG%\Classes\ADOC\:

Specialties

There are some special cases we need to discuss.

Inheritance

With version 2.0, ADOC has learned to deal with inheritance. That means that inherited classes are contained in the documentation by default. Inherited members are marked as such.

You can suppress this default behaviour by setting the "Inherit".

:Include

With an :Include statement one or more namespaces, either ordinary ones or scripted ones, can be included into a class. Since it is possible for a function to carry a line with

:Access Public

in a namespace, this makes it possible to enhance the public interface of a class script. Since version 1.3, ADOC is able to deal with this: included public methods are reported correctly: in the documentation they appear like ordinary methods.

Embedded Classes

There are good reasons to keep a class as small as possible but sometimes it is important to make a class self-contained to keep it independent from other scripts. If a class contains lots of stuff dealing with files, or the Windows Registry, it makes perfectly sense to organize them into classes contained in the master class. Such a class is then called an “embedded class”.

If you are looking for what a class can do for you, you are not interested in any embedded classes: They are part of the implementation details, and normally you are not interested in those details.

However, if you need to change or enhance a class which carries one or more sub-classes, things are quite different. In this case, you might be interested in the public interface of the embedded classes, because you are supposed to use them.

With

```
parms←⎕ns''
parms.ReportEmbeddedClasses←1
parms #.ADOC.Browse #.ADOC
```

you get a documentation all classes embedded in #.ADOC.

Note that there is a method which can be used to get a list of all embedded classes:

```
#.ADOC.ReportEmbeddedClasses #.ADOC
```

Misc

Copyright

You can display the copyright notice by calling the `ADOC.Copyright` method. The software is under the so-called MIT license. For details, look at

http://en.wikipedia.org/wiki/MIT_license

Latest version

The latest version can be downloaded from
http://aplwiki.aplteam.com/moin.cgi/AplDocDownloadPage

You can get version information by calling:

```
ADOC.Version
```

Enhancements and New Versions

The basic forum for ADOC is the APL wiki:

http://aplwiki.aplteam.com/

ADOC is part of the TOSAP project: "The Open Source APL Project". On the wiki, you will find all kind of information associated with classes' part of TOSAP.

If you are willing to contribute, look at

http://aplwiki.aplteam.com/moin.cgi/OpenSourceApl

for details how to do this.

Bugs

As any TOSAP project, ADOC is owned by somebody. You can find out by inspecting the bottom part of ADOC's homepage in the APL wiki.

In case of a bug, send a description to the owner that makes it as easy as possible to reproduce the problem.