JWSCL Coding Conventions
Version 1.0

Christian Wimmer

July 7, 2008

Contents

1 Who should read this?

This documentation is for all people who want or are writing code for the JEDI Windows Security API project. It is intended to help them to improve the source code, the readability and the documentation .Try to stick as much as possible and ask your tutor or JEDI Administrators if there are problems, additions and more.

2 General

Before you start writing your JEDI code read this documentation completely. It avoids additional work to be done in your code. You should also read the JEDI style guide conventions located in the documentation file "'JEDI StyleGuid.pdf"' located in the subversion folder jwscl/trunk/documentation[1] on your harddisk. Try to stick to the styleguide as best as possible.

3 Contact

You can contact us at mail@delphi-jedi.net.

As a JEDI API&WSCL member you can contact us using the instant messanger protocol Jabber.
The JEDI Jabber Accounts of the current Administrators (July 2008) are:

- **Christian Wimmer** christianwimmer@delphi-jedi.net
- **Remko Weijne**n rweijnen@delphi-jedi.net

The JEDI Jabber Server cannot be contacted through other Jabber Servers. You need an account on the server. The account can only be received by an invitation!
The logon credentials are:
t erver
delphi-jedi.net
Port
30000
Activate
SSL!

4 Target Delphi Versions

- **JWSCL** can be compiled with **Delphi 7** and newer. Do not use new Delphi statements without using compiler IFDEFS defined in jedi.inc. This include file is automatically included by jwscl.inc which is used by all JWSCL units. For example you can check for Delphi 2007 using **DELPHI11_UP**.

[1] It can be downloaded from http://jedi-apilib.svn.sourceforge.net/viewvc/*checkout*/jedi-apilib/jwscl/trunk/documentation/JEDI StyleGuide.pdf?revision=4

```
{$IFFDEF DELPHI11_UP}
 new Delphi code
{$ELSE}
 backwards compatible code
{$ENDIF DELPHI11_UP}
{$...} //don't use new compiler directives outside!
```

- If you want to use a new Delphi construct you must make sure that a programmer with an old Delphi version has also the possibility to use the feature in another way.
- New Delphi versions (since Delphi 2005 to be exact) implements code region. This region can be used to show or to hide code areas in the Delphi IDE. It has no other purpose. This feature is problematic in JWSCL units since it makes it impossible to compile the project. Of course it is possible to use this feature in examples or projects that includes JWSCL. However in this way the example/project is limited to the new Delphi versions.

```
{$REGION 'name'}
folded code
{$ENDREGION}
```

- There is no FreePascal support at the moment.

5 Source Formatter

It is possible to use a source formatter. However don't use a source formatter that uses a format that differs too much from the existing source format. An appropriate formatter has the JEDI Code Format Project[2].

6 Configuration Managment (with Subversion)

The JEDI WSCL Project and its sister project the JEDI API is maintained with Subversion hosted on Sourceforge.net. It can be downloaded using a Subversion Client[3]

6.1 Layout

The Subversion repository is structured into *tags*, *branches* and a *trunk* folder.

- **Trunk** The trunk folder is the main developer folder. It includes code that has alpha state and should used with care. The following sub folders are available:

[2]The JEDI Code formatter can be found at http://jedicodeformat.sourceforge.net

[3]Version 1.4.6 command line client: http://subversion.tigris.org/files/documents/15/41687/svn-1.4.6-setup.exe
Windows Client: http://tortoisesvn.tigris.org

  - **COM** This folder contains sources that implements (nearly) all JWSCL classes as COM interfaces. In this way it will be possible to access the JWSCL through C++, Basic, script languages and many more. However this endeavour isn't finished yet and thus the code is not usable yet.
  - **documentation** This folder contains documentation about the JWSCL. It also contains the JEDI StyleGuide.
  - **examples** This folder contains examples which show how to use the library.
  - **packages** This folder will contain packages that are used to compile all JWSCL unit. In this way it is possible to automatically do syntax checking.
  - **source** This folder contains the main source code files of JWSCL. Each delphi file is accompanied by a dtx file which contains external documentation for Doc-O-Matic. It will be created and maintained only by the document managment.
  - **unittests** This folder contains tests for (nearly) all classes of JWSCL. Of course many classes cannot be tested because of their nature. So it is hardly possible to test a multi thread environment or windows API calls that depend on the environment - in this way.

- **Branches** This folder contains a copy of the Trunk folder at a specific version. In this way it is possible to create a parallel development to do specific changes or releases. A release has a version number identifier (like 0.9.1). A release is not used to add additional features but just bug fixes.

- **Tags** This folder contains a static copy of the Trunk or Branch version of the code at a specific revision[4]. In contrary to the Branches folder, a tagged version cannot be changed[5].

6.2 A new release

When a new release it about to be published and all features of a trunk are ready to use, a new branch is created with the version identifier (e.g. 0.9.2) and a pre-release tag "'b"' (which stands for beta like 0.9.2b). In this way the release can be tested and finally be released without the beta tag. The upcoming release is feature-complete and thus will only receive fixed bugs.

6.3 Commit

Always make sure that your code can be compiled properly without errors and to come off best no warnings at all. It is always an annoyance to get errors and to wait for a fix.

[4] A revision is a code version created at a commit.

[5] Infact the Subversion server does not deny write access to the Tag folder. However it is against the rules to change the Tags.

6.4 Risk

Committing source code can sometimes be problematic or even dangerous. You should always make sure that you don't include sensible information (like passwords) into the source code and upload it to the server. Everybody can download the source code and thus get your credential data. If it is too late the only choice you have is to change your login data.

7 What belongs where?

- Add your exceptions to *JwsclExceptions.pas*.
- Add your simple types to *JwsclTypes.pas*.
- Add your constants to *JwsclConstants.pas*.
- Add utility functions to *JwsclUtils.pas*.
- Add COM utilities to *JwsclComUtils.pas*.

8 Naming conventions

- Name your classes with the JEDI Windows lettes **Jw**

```
TJwYourClass = class
```

- Start every word of a name with a capital letter

```
ThisCanBeAName
```

- Constant names are to be written in capital letters

9 Using Strings

- Use *TJwString* and *TJwPChar* if you support Ansi- and Unicode. Otherwise use *AnsiString*, *WideString* and *PAnsiChar* and *PWideChar* instead of simple *String* and *PChar*. The last two simple types may become incompatible with newer Delphi versions[6].
- If you need to call a WinAPI function that is Ansi- and Unicode aware you must implement them both instead of calling the generic function [7].

[6] It is directly addressed to the upcoming Delphi which implements Unicode by default. So String will automatically become WideString. This may break existing JWSCL code.

[7] The JEDI Windows API implements automatic creation of Ansi- or Unicode functions. Most times a Windows API function is implemented three times. E.g. CreateProcessA (Ansicode), *CreateProcessW* (Unicode) and *CreateProcess* that points to *CreateProcessA* or *CreateProcessW* depending on the compiler directive **UNICODE**. If it is set *CreateProcessW* is used; otherwise *CreateProcessA*. However JWSCL does only rely on its on **UNICODE** directive.

```
If {$IFDEF UNICODE}CreateProcessW{$ELSE}
    CreateProcessA{$ENDIF UNICODE}(TJwPChar(Name)
    ,...) then ...
```

You can use TJwPChar to automatically cast to PWideChar or PAnsiChar depending on the UNICODE directive.

- Don't add strings directly into the source code. Instead add string constants to JwsclResource.

10 Exceptions

- Use the whole power of the exception constructors. An example for a failed windows api function call can look like

```
raise EJwsclWinCallFailedException.CreateFmtWinCall(
 //Message as a resource string
 RsSecurityDescriptorInvalid,

 //Source procedure name
 'GetJobObjectInformationLength',

 //Source class name
 ClassName,

 //Source unit file name
 RsUNSid,

 //Source line (set to 0)
 0,

 //add GetLastError information?
 True,

 //Windows API function name that failed
 'QueryInformationJobObject',

 //format string parameters for message
 ['...']);
```

- Instead of returning an error code, always raise an exception.
- Create your own meaningful exception classes derived from EJwsclSecurityException or any of it descendants.
- Never use the *EJwsclSecurityException* in a **raise** statement.
- Do not use WinAPI calls in a EJWsclSecurityException (or derived) constructor call since it may change the GetLastError value before the constructor can read it. Instead save the GetLastError manually in a variable and reset it by calling SetLastError.

```
var
  StoredLastError : DWORD;
  DataString : AnsiString;
begin
  if not <WinAPI Function> then
  begin
    StoredLastError := GetLastError;
    DataString := <Another WinAPI Function>;
    SetLastError(StoredLastError);
    raise EJwsclWinCallFailedException.Create(...,
        DataString, ...);
  end;
end;
```

- Try to check for specific exceptions that you can handle. Do not catch generic exceptions like *Exception* and *EJwsclSecurityException*.

11 Create your own JWSCL unit

If you are going to create your own JEDI Windows Security Code Library Unit you should read the following sections. There is a template unit "_JwsclTemplate.pas" that you can use for your first steps. Make a copy of the file, rename it, adapt the unit name in the source code and start coding. Read on with the following sections.

11.1 Header documentation

The template unit contains some place holders that you should replace with the corresponding values

1. **<Description here>** Add the description of the provided features of this unit.
2. **<Author name>** Add all authors who supplied work to the unit.
3. **<UnitName>** Replace with the correct unit name.

You also should add additional information about the unit if necessary. It may contain

- A more detailed description.
- A list of known bugs.
- Example(s) how to use certain features.

11.2 Names

Make sure that you name your identifiers are unique in all JWSCL units. Thus you should add resource strings, constants and simple types to special units as described in 7.

12 Documentation

Documentation is a main part of the JWSCL. There are two types of documentation:

1. **internal comments** Internal comments are used to describe difficult or problematic source code parts. The used language is simple english.
2. **declaration comments** Declaration comments are comments that describes a class, a method, a function, a constant or a simple type. This type of comment uses a particular syntax, called **JavaDoc**[8] and **XMLDoc**[9].

12.1 Documentation rules

- Document your code if it is problematic to understand
- Use only "{" and "}" to document your code. It is also possible to use "//" to comment out only one or a few lines. In this way "(*" and "*)" can be used to comment out parts with different comments temporarily.
- Document all constants, functions, procedures, types, classes, protected, public and published methods and properties. It is not necessary to document private parts that are only helper methods or variables for properties (which are already documented).
- Use XMLDoc and/or JavaDoc documentation style. The help creator *Doc-O-Matic* is used to grab all documentation and to create a documentation file. Example
  More examples can be found at the Doc-O-Matic homepage. Be aware that the JWSCL doc headers may differ from the shown examples because they were translated from another (incompatible) comment style.

```
{
<B>IsProcessInJob</B> returns whether a process is
    assigned to the job or not.
@param hProcess defines any handle to the process that
    is tested for membership.
@param Returns tre if the process is a member of the
    job; otherwise false.
@return Returns true if the given process is assigned
    to the current job instance; otherwise false.
raises
  EJwsclWinCallFailedException: can be raised if the
      call to an winapi function failed.
}
function IsProcessInJob(hProcess : TJwProcessHandle) :
    Boolean;
```

- Add all exceptions that can be raised into the documentation even if they may be raised in used methods. In the last case you can list all used methods instead so the user can see which exception may be raised additionally.

[8] http://java.sun.com/j2se/javadoc/writingdoccomments/index.html
[9] http://msdn.microsoft.com/de-de/magazine/cc302121(en-us).aspx

13 Implementation

- Don't use private declarations if there isn't a very good reason. First ask your tutor or Administrator. The reason is that derived classes should be allowed to acces the internal variables. It is the programmers choice to decide how to protect internal variables.

- Stick to this class example

```
type
  {to be documented}
  TJwMyClass = class(...)  //see Jw ?
  protected    //we use protected, so derived classes
      can use it
    fVariable : Integer; //see f the and big letter?

    {you could add documentation here
    but it is not necessary since it is described in
        property}
    procedure SetVariable(const Value : Integer);
  public
    {to be documented}
    constructor Create...
    {to be documented}
    destructor Destroy; override;

    {to be documented}
    property Variable : Integer read fVariable write
        SetVariable;
  end;
```

- Always use try finally to free allocated memory. The following listing shows the allocation of an untyped pointer and a class instance. Be aware of the use of two try/finally statements. This is necessary because the status of the second allocation is undefined at first and thus must not be freed.

```
var
  Ptr : Pointer;
  MyClass : TMyClass;
begin
  GetMem(Ptr, 100);
  try
    //call to functions which may raise exceptions
    try
      //call to functions which may raise exceptions
      MyClass := TMyClass.Create;
      //call to functions which may raise exceptions
    finally
      MyClass.Free;
    end;
  finally
    FreeMem(Ptr);
  end;
```

```
end;
```

- It is not necessary to set both variables (in the last listing) to nil because these variables are never used again. However if you are going to implement these variables globally, you should use instead the procedure **FreeAndNil** or set to **nil**.

- Instead you can also use *TJwAutoPointer.Wrap* method which returns an *IJwAutoPointer* interface. Such a "wrapped" pointer, handle or instance is automatically destroyed as soon it goes out of scope[10] and even if an exception forces it out of scope.

  The previous listing can be rearranged using TJwAutoPointer.

```
var
  Ptr : Pointer;
  MyClass : TMyClass;
begin
  GetMem(Ptr, 100);
  TJwAutoPointer.Wrap(Ptr, ptGetMem);

  MyClass := TMyClass.Create;
  TJwAutoPointer.Wrap(MyClass);

  //call to functions which may raise exceptions
  raised Exception.Create(''); //test exception
end;
```

  Both pointers are automatically destroyed if the instruction pointer reaches the **end;** statement. It even happens if an exception is raised. If you want wrap data which should live longer as the ones in the above example you have to store the interface returned by the method *Wrap*. It is even possible to access the data through the interface. Thus there is no need to use a second variable. However the interface only contains an untyped pointer, TObject and handle.

```
type
  TMyClass = class
  protected
    fMyPointer : IJWAutoPointer;
    fMyClass : IJWAutoPointer;

    constructor Create;
    procedure Foo;
  end;

constructor TMyClass.Create;
var
  Ptr : Pointer;
```

[10] A scope is an area where a variable can be access. If a wrapped variable is created on stack it will be removed as soon as the last "end;" statement of a method is reached. Furthermore a wrapped variable in a class is freed as soon as the instance is destroyed. And at last a globally defined wrapped variable is freed when the application exists.

```
  MyClass : TMyClass;
begin
  GetMem(Ptr, 100);
  fMyPointer := TJwAutoPointer.Wrap(Ptr, ptGetMem);

  MyClass := TMyClass.Create;
  fMyClass := TJwAutoPointer.Wrap(MyClass);
end;

procedure TMyClass.Foo;
var
  Ptr : Pointer;
  MyClass : TMyClass;
begin
  Ptr := fMyPointer.GetPointer;
  MyClass := fMyClass.GetInstance;

  //do more
end;
```

Of course you must be aware that typed pointers should not be converted to untyped pointers since it is hard to understand and difficult to find errors.

- Use the parameter directive **const** as often as possible. It makes possible to improve speed and comprehensibility of parameters. A constant parameter cannot be changed within the function and thus the compiler can create a call by reference instead of a call by value (which copies the the data). Of course it does not apply to the internal members of the parameter if the parameter is a class for instance. A user of the function can use the original data with the fear of changing it.