

JavaExe

version 3.1



by DevWizard
(DevWizard@free.fr)

(6 June 2012)

to Jawaher...

Table of Contents

Presentation.....	7
History.....	9
General Use.....	11
Creation of .EXE.....	11
Properties	12
Use of UpdateRsrcJavaExe.....	14
Change the icon of .EXE	15
Splash Screen	15
Running as Windows application	17
Splash Screen	17
Instances Number.....	17
Session Restore	17
Running as Windows Service	19
Running as Control Panel	23
Additional functionalities	
System Events Management	27
Taskbar Management	31
Windows Registry Management	35
Appendices	
Java Interfaces.....	41
ApplicationManagement	41
ControlPanelManagement.....	41
RegistryManagement	42
ServiceManagement.....	42
SystemEventManager.....	43
TaskbarManagement.....	44
Examples.....	47
1 - Application.....	47
2 - Control Panel	47
3 - Service	47
4 - TrayIcon.....	47
5 - Service & TrayIcon.....	48
6 - System Event	48
7 - OneInstance	48
8 - Service & TrayIcon & System Event	48
9 - Registry	48
10 - Test Unicode	49
11 - Restore Session.....	49

Presentation

JavaExe makes it possible to launch your Java application from an .exe file like a Windows application, or a service system or as a Control Panel.

It is possible to provide a JRE with the Java application so that it operates regardless of the configuration of the client system.

Among the features of **JavaExe**, besides the launch as a Service or Control Panel, we can limiting the number of instance running, the restoration of the Java application automatically after a system reboot, the interception of Windows systems events (such as inserting or ejecting an external device, a reboot request of system and allow it or not, change of state of the user session, connection or disconnection of a network, change of state of the battery, ...), or a feature to manage the taskbar or managing of the Windows registry.

« Imagination is more important than Knowledge »

Albert Einstein

History

- **3.1 (6 June 2012) :**
 - Unicode : Full management of Unicode in **JavaExe** (except for the class name).
 - Windows Registry : integration of native functions are accessible from Java applications.
 - Possibility to automatically restart the Java application after a reboot.
 - Taskbar : Adding the mouse click event on the balloon.
 - Windows Services :
 - Adding of failure actions (RESTART, REBOOT), and Delayed Automatic starting.
 - Changing the launch of the interactive part of the service.
 - Control Panel : Fixed a major bug in the installation of the control panel from a Windows Vista and higher.
 - Fixed a minor bug when the JRE is included with the application and that it runs on a Windows empty. The JVM could not find the MSVCR71.dll file.
 - Fixed a minor bug on the total size of arguments passed to the executable file of the Java application (**JavaExe.exe** renamed).
 - For launch as a service or control panel, the current path is fixed to the Java application, where is the executable (instead of "C:\WINDOWS\system32\" by default).
- **3.0.2 (14 February 2007) :**
 - Correction of minor bug with JRE 1.4 : When the Java application exits with a *System.exit(0)* an error file was generated by the JVM. This error occurs only with JRE 1.4.
 - Correction of major bug with JRE 1.6 : When the JRE 1.6 was provided with the Java application, **JavaExe** could not find the main class.
- **3.0.1 (30 October 2006) :**
 - Correction of a minor bug in **UpdateRsrcJavaExe** : the files associated with the checkbox were always taken even if the corresponding checkbox were not selected.
 - Correction of a major bug concerning the example "7 - OneInstance": the result of the method *isOneInstance* was not always taken in certain version of Windows XP, and the example "8 - Service & TrayIcon & System Event" : the interactive part can't run in some cases.
 - The minimum number of version required of Java is indicated in the message of alert if no JRE is found.
- **3.0 (11 September 2006) :**
 - Management of Java application as a Control Panel (with **JavaExe.cpl** file)
 - Management of the Taskbar (Icon Tray).
 - Management of System Events.
 - Displaying a SplashScreen while starting the application.
 - Possibility of controlling the number of instances running of same application.
 - Rename the tool **MergeICO** into **UpdateRsrcJavaExe**.
 - Properties :
 - Add *URL_InstallJRE*, *PathJRE*, *PathBrowser*, *Display_BoxInstall*
 - RunAsService* : rename to *RunType*
 - RunType* : new type adding (2) for the Control Panel mode.
 - ClassDirectory* : setting to default to "resource"
 - Reading "manifest" of main .jar to looking for the main class.
- **2.0 (16 November 2003)**
 - Launching the Java application directly with the JVM if possible. If not, launching it via **java.exe**
 - Possibility of launching the application like a Windows service.

- Creation of second executable file named **JavaExe_console.exe** for launching the application with a console DOS.
- Addition of some properties: *ClassDirectory*, *PersonalOptions*, *ResourceDirectory*, *RunAsService*
- The *JREversion* property means now the minimum version instead of the strict version.
- **1.3** (21 April 2003) :
 - Correction of a potential bug in **JavaExe.exe** (a pb with “\” in the variable properties *PersonalClasspath*)
- **1.2** (4 November 2002) :
 - Correction of a bug in **MergeICO.exe** (the moving of an icon file to **MergeICO.exe** didn’t work)
 - Launching of the Java application with the parameter *java.library.path* fixed at “;\resource\”, you are thus allowing to put your possible DLL (for the native methods for example) in the same directory as your application or in the directory “resource”.
- **1.1** (5 October 2002) :
 - Addition of a property, **Main Class**, in the **JavaExe.properties** file. This property is necessary when the main class is in a package.
- **1.0** (28 August 2002): initial version.

General Use

Creation of .EXE

To obtain an .exe file of your Java application, it is quite simply enough to copy **JavaExe.exe** in your directory containing the Java application, then to give him the same name as your principal .class or .jar. **JavaExe.exe** is provided with a console version, **JavaExe_console.exe**, to allow have a console DOS to print on standard output. All that will be said on **JavaExe.exe** apply to **JavaExe_console.exe**.

Example :

If my main class names *MyApp.class*, I copy and rename **JavaExe.exe** to **MyApp.exe**

If my principal class is contained in a .jar, this one will have to be also called *MyApp.jar*.

The .class or .jar must be in the same directory as the .exe or in a directory named by default “resource” to create on the same level as the .exe file. However this directory can be defined specifically by modifying the property “**ResourceDirectory**” (see the paragraph named *The Properties*).

Example :

If **MyApp.exe** is in the directory “**D:\Dev**”, then *MyApp.class* or *MyApp.jar* is in :

- either in “**D:\Dev**”
- either in the directory “**D:\Dev\resource**”

JavaExe remains however dependent on a JDK or a JRE, it is necessary that at least a Java Runtime Environment (JRE) is installed. If **JavaExe** does not detect a JDK or JRE, it will open a browser on the site of Sun to download the current JRE.

You can provide a JRE with your application (the completely uncompact JRE and not the installation file). In this case, you must put it in a directory named « jre », itself in the directory of the .EXE or in the directory “resource”.

Example :

Let be the following configuration of **MyApp.exe** :

- the .exe is in the “**D:\Dev**”
- a JRE is provide with the application and is in the directory “**D:\Dev\resource\jre**”

Then *MyApp.exe* will always launch with this JRE there some is that installed on the machine customer, even if it of installed of it there none.

Properties

Once the .exe file created, it is possible to associate properties it to define the way in which the Java application will be launched or to specify certain parameters necessary to its executing.

These properties are to be put in a text file bearing the same name as the .exe file, but with the extension “properties”. A property will be defined by a followed name of its value, form: “name = value”.

However this file could be integrated into the .exe by using the **UpdateRsrcJavaExe** utility.

Example :

If **MyApp.exe** is in the directory “D:\Dev\”, then **MyApp.properties** can be in this same directory or in “D:\Dev\resource\”.

In this example, **MyApp.properties** contains:

```
JRE version = 1.2
Personal Classpath = .\resource\classes12.zip
MainArgs = "test" 123
```

MyApp will be then launched with Java 1.2 (or more), and orders it command line is :

```
java -classpath .;.\resource\MyApp.jar;.\resource\classes12.zip MyApp "test" 123
```

Here the list of these properties :

- **JRE version** (or **JREversion**) = to specify a minimal version of java : **1.4 ; 1.3 ; ...**

If a JRE is provided with the application, this property will be ignored.

example :

JREversion = 1.3 *JavaExe must be able to find at least version 1.3 of Java to launch the application*

- **Run Type** (or **RunType**) = to specify how the application must be launched :

0 = as a simple application (default value)

1 = as a service

2 = as a Control Panel

example :

RunType = 1 *JavaExe will launch the application as Service*

- **Run As Service** (or **RunAsService**) = this property should not be used any more. To replace by “RunType = 0” or “RunType = 1”

- **Main Class** (or **MainClass**) = to indicate the complete name of your main class, if JavaExe could not find it according to only the name of .exe or Manifest in the jar. The only case where it is necessary to specify this property will be when the name of the .exe and the .jar does not reflect the name of the main class and no Manifest is found.

example :

MainClass = com.toto.myClass

- **Main Args** (or **MainArgs**) = these values will have passed in arguments to the method main of your principal class, in the variable (String[] args).

example :

MainArgs = 123 aze *the argument args[] of the method main will contain: [0] = “123” and*

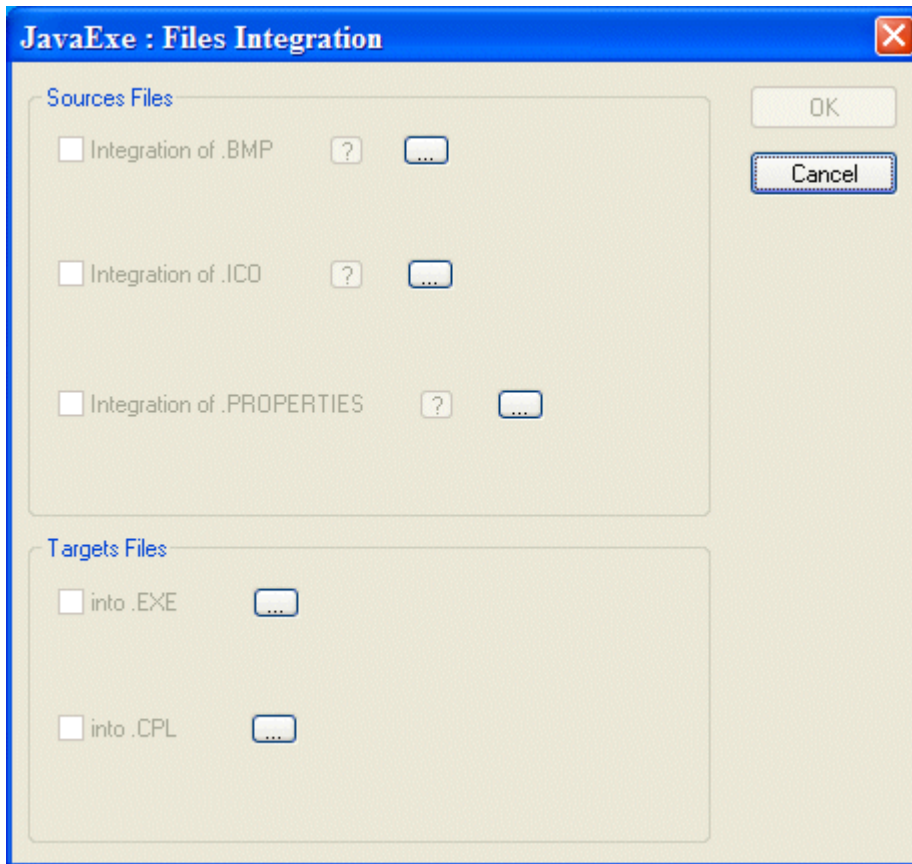
[1] = "aze".

- **Personal Options** (or **PersonalOptions**) = allows to specify the options of launching specific to the JVM.
example :
PersonalOptions = -Xms64m -Xverify:none
- **Personal Classpath** (or **PersonalClasspath**) = if your application to need for .jar, .zip or .class additional or being in other directories. Several files or directories can be specified by separating them from a semicolon.
example :
PersonalClasspath = D:\Dev\lib\lib.jar ; C:\Application\resource
- **Resource Directory** (or **ResourceDirectory**) = to indicate the directory resource containing the .JAR, the .DLL, the images, files of properties.... If this parameter misses, the directory named "resource" located at the same level that the .EXE will be used by default.
example :
ResourceDirectory = .\bin specify this directory where the main .jar must be sought by default.
- **Class Directory** (or **ClassDirectory**) = to indicate the directories (separated by `;') to scanner recursively in order to find all there .jar and .zip to be put in ClassPath. This property will contain at least the directory "resource" thus allowing the taking into account of all the .jar contained in this directory without having to specify them one by one in the classpath.
example :
ClassDirectory = .\lib\ ; D:\Dev\lib add to ClassPath all the .jar and .zip found in these 2 respective directories and their sub-directories, like in the directory "resource".
- **Path JRE** (or **PathJRE**) = pathname of the JRE if it is provided with the application. By default it will be required in the directory "jre" on the same level as the .exe or in the directory "resource".
- **Path Browser** (or **PathBrowser**) = pathname of the browser to be used for the installation possible of a JRE (by default it is the pathname of InternetExplorer).
- **Display BoxInstall** (or **Display_BoxInstall**) = to indicate if a message must be displayed when JavaExe does not find a JRE or JDK, and asking whether one wishes to install a JRE or to leave the application. Only two values are accepted : 0 or 1.
1 = display dialog box to install or not the JRE (default value)
0 = do not display any message, and starting the procedure of installation by opening a browser on the URL adequate.
- **URL InstallJRE** (or **URL_InstallJRE**) = allows to indicate a URL on which JavaExe will open a browser if no JRE or JDK is found with the launching of the application. If this property is not indicated, it is the URL on java.sun.com which will be taken.


There can be other properties if your application uses this same file for its own needs.

Use of UpdateRsrcJavaExe

JavaExe is provided with another program, **UpdateRsrcJavaExe** (called **MergeICO** in the previous versions), making it possible to change the icon of your **MyApp.exe**, to define a splash screen, or to integrate the file of the properties in the .exe or the .cpl (for the use of the Java application as Control Panel).






The integration of these files can be done in three ways :

- While clicking on the button  of the type of file which one wishes to open.
- By moving the files wanted on this window of **UpdateRsrcJavaExe**.
- By command line.

Types of recognized files :

- **.BMP** : allows to define a splash screen in the Java application.
- **.ICO** : allows to change the icon of the .exe file.
- **.PROPERTIES** : allows to integrate the properties used by JavaExe.
- **.EXE** : allows to specify the .exe derived from JavaExe.exe (renamed or not) which will receive the files to be integrated.
- **.CPL** : allows to specify the .cpl derived from JavaExe.cpl (renamed or not) which will receive the files to be integrated (only the file of properties can be integrated into a .cpl).

After loading a file to be integrated, it is possible to see the characteristics of them while clicking on its button  .

When at least a source file and a destination file are loaded in **UpdateRsrcJavaExe**, it will be then possible to click on the button  to execute the integration of the files whose box  will be checked.

If **UpdateRsrcJavaExe** is used in command line, here the list of the recognized arguments :

- **-run** : allows to launch integration without the window require to open if all the parameters necessary are specified.
- **-exe=file** : to indicate the name of a .exe file which will receive the files to be integrated. This .exe file must be a derived of JavaExe.exe.
- **-cpl=file** : to indicate the name of a .cpl file which will receive the files to be integrated. This file must be a derived of JavaExe.cpl.
- **-ico=file** : allows to indicate the name of an icon which will be integrated into the .exe.
- **-bmp=file** : allows to indicate the name of a BMP image which will be integrated into the .exe and being used as splash screen.
- **-prp=file** : to specify the name of a properties file which will be integrated into the .exe and the .cpl.

To change the icon of .EXE

It is possible to modify the icon of the .exe file to launch your Java application. All formats of icons are now accepted by **JavaExe**.

With this intention it is enough to use **UpdateRsrcJavaExe** provided with **JavaExe**. (see the preceding paragraph for its use).

Splash Screen

To define a splash screen in your Java application it is enough to have the image in format BMP and to use the **UpdateRsrcJavaExe** program. (see the paragraph of this utility).

Running as Windows application

To launch your Java program as a Windows application, you do not have anything special to make if it is not what already was note in the chapter “General Use” : Just rename **JavaExe.exe** giving the same name as your main .class or .jar.

Splash Screen

If the Java application is launched with a splash screen (see the previous chapter), it is possible to control the time of appearance of this. For that it is enough to define the following static method in the main class :

```
public static boolean isCloseSplash();
```

As long as this method returns **FALSE**, the splash screen will remain visible. If it returns **TRUE**, the screen disappears and the method will not be called any more.

If this method is not defined, the splash screen will remain some time, fixed by **JavaExe**.

Instances Number

It is possible to control the number of instance of the Java application, while authorizing or not only one execution at time. For that your main class must contain a named static method “*isOneInstance*” and must have the following signature :

```
public static boolean isOneInstance (String[] args);
```

The arguments sent to this method are those which will be sent to the *main* method. If *isOneInstance* returns **TRUE** then only one instance of the application will be launched.

At the time of the launching of the application, if it is the first instance that executing, this method will not be called but the *main* method with its possible arguments.

On the other hand, if it is not the first execution, the method *isOneInstance* of the first instance of the application will be initially called with the arguments which the *main* method would have received.

If *isOneInstance* returns **TRUE** the process stops there and the launching instance will be cancelled. If *isOneInstance* returns **FALSE** the process of launching continues, a new instance of the application will be executed and its *main* method will be called with the possible arguments.

Session Restore

During a system reboot if the Java application was running, it can tell to **JavaExe** whether to store the current context of the application to return it by restarting the application automatically with the system.

The kept context correspond to the arguments passed to the application and the session data supplied by the application. To manage this session restore, just set the following static methods in the main class :

public static *boolean* **sessionIsRestore()**;

The declaration of this method is optional. It used to tell JavaExe restarting or not the application after rebooting the system.

If it returns TRUE, the application will be restarted even if no context isn't given by methods **sessionGetMainArgs()** and **sessionGetData()**.

However if the method returns FALSE, the application will not be restarted regardless of the statement or values of the two methods mentioned above.

Finally, if the method is not declared, the application will be restarted if at least one of the two previous methods returns a value.

public static *String[]* **sessionGetMainArgs()**;

This method is optional and provided to **JavaExe** additional arguments to be passed to the main method when the application is restarted. These arguments are added to those existing when the application was launched with arguments.

public static *Serializable* **sessionGetData()**;

This method, if it is declared, provides to **JavaExe** some data that will be returned to the Java application after being relaunched with the system. This maintains a state of the application while Windows restarts.

public static *void* **sessionSetData** (*Serializable* data);

This method is called automatically by **JavaExe** after the application has been restarted and before calling the main method, with the context data provided by the method **sessionGetData()**.

Running as Windows Service

So that your Java application is launched as a service system, it is enough to create the .exe (see the chapter “General Use”) and to specify in the file properties, the property “RunType = 1”.

A restriction should however be noted : the service will not be able to launch in console mode with **JavaExe_console**.

With the launching of the application several cases of figure can exists :

1. the main class is provided to function like a normal application, i.e. the point of entry is **main()**.
2. the Java application contains the definite methods for **JavaExe** being used as interface between the management of the Windows service and the application (see lower, like in Appendix the **JavaExe_I_ServiceManagement** interface).

And for each one of these cases, the application-service can be launched directly with the JVM or via the command **java.exe**. That thus makes us 4 cases of launching be studied.

1. **main() + JVM** => the point of entry being **main()**, this one will be called only to launch the service, and this last could be stopped only by restarting the system.
2. **main() + java.exe** => same thing that previously.
3. **interface + JVM** => the defined methods to be used as interface will be called individually according to needs'. The method **main()** will never be called.
4. **interface + java.exe** => since launching is executing with the command **java.exe**, the point of entry will be then **main()** and we fall down in the configuration of the case n° 2.

In the case n° 3, if for an unspecified reason one cannot call the JVM directly, one will have to pass by **java.exe** (case n°4) and thus the method **main()** will be the only point of entry. Also, it is important not to forget to call the method **serviceInit()** since **main()**. For more details to see the example provided with this documentation.

It is possible to directly launch operations on the service, like its installation, its suppression, its start or its stop, without passing by the possible dialog box of confirmation.

For that it is enough to launch **JavaExe.exe** (i.e. **MyApp.exe**) with like argument :

-installService	: to force its installation
-deleteService	: to force its suppression
-startService	: to force its start
-stopService	: to force its stop

Methods used as interface : JavaExe_I_ServiceManagement

These methods are directly called by **JavaExe** :

```
public static boolean serviceIsCreate ();
public static boolean serviceIsLaunch ();
public static boolean serviceIsDelete ();

public static boolean serviceControl_Pause ();
public static boolean serviceControl_Continue ();
public static boolean serviceControl_Stop ();
```

```

public static boolean serviceControl_Shutdown ();

public static String[] serviceGetInfo ();
public static boolean serviceInit ();
public static void serviceFinish ();

public static void serviceDataFromUI (Serializable data);
public static boolean serviceIsDataForUI ();
public static Serializable serviceDataForUI ();

```

These methods are to be declared either in the main class, or in a class with the same name but post fixed by “_ServiceManagement”. For example, if my main class is called *MyApp*, then these methods can be indifferently in *MyApp.class* or *MyApp_ServiceManagement.class*.

It is not necessary to declare all them.

1. **serviceIsCreate** : This method is called at launching of *JavaExe.exe* (i.e. *MyApp.exe*) if the service is not installed yet. The service will be installed only if this method returns TRUE. If this method is not declared, a dialog box will open to require of the user if it or not wishes to install the service. The method **serviceGetInfo** will be also called to obtain certain characteristics necessary to the creation of the service.
2. **serviceIsLaunch** : This method is called after the installation of the service. This one will be immediately launched if the method returns TRUE. A dialog box will open, if this method is not declared, to require of the user if it or not wishes to launch the service.
3. **serviceIsDelete** : This method will be called with launching of *JavaExe.exe* (i.e. *MyApp.exe*) if the service is already installed. The service will be removed only if this method returns TRUE. If this method is not declared, a dialog box will open to ask whether the user or not wishes to remove the service. However if the service were created by specifying that its stop was not authorized (see the method **serviceGetInfo**), the service will be actually removed only with the restarting of the system.
4. **serviceInit** : This method is called when the service is launched, whether it is manually or automatically. The method must return TRUE if and only if the application is active and in executing. If it returns FALSE or if it does not answer before a 30 seconds deadline, Windows will consider that the service failed the attempt at starting and will launch the program of failure then if it were defined (see the method **serviceGetInfo**). If the method is not declared, the service will be launched immediately without condition.
5. **serviceFinish** : This method will be called when the service is stopped either manually, or automatically with the stop of the system.
6. **serviceGetInfo** : This method is called at the time of creation of the service in order to obtain certain additional information, such as :
 - Complete name of the service in opposition to the short name which is the name of the .exe file.
 - Service Description.
 - “1” or “TRUE” to indicate that the service will be launched automatically with the system.
 - “1” or “TRUE” to indicate that the service can be stopped manually.
 - Name of the file to be executing when the service failed. Files .BAT can not be executing correctly on Windows 2000.
 - Arguments required to the program which is executing at the time of a failure.
 - List names of services (short name), separated by a tabulation ('\t') or slash ('/'), on which this service depends. i.e. Windows will make sure that these services are launched before launching this one.
 - List of actions for service failure. The possible values are: NONE, RESTART, REBOOT or RUN corresponding to "Do Nothing", "Restart Service", "Reboot System" or "Run Program". This list can contain multiple values separated by a slash (/). For example: RESTART / RESTART /

REBOOT, the system will restart the service for the 1st and 2nd failures and restart Windows for the 3rd failure. The number of values is not limited but Windows will only display the first 3. However all values in the list will be taken into account by the system.

- List of delay (in seconds, and separated by a slash '/') corresponding to the actions to be triggered on failure. This list must contains same number of values that the action list. "10 / 20 / 30" for example corresponds to an expectation of 10 seconds before triggering the first action, then a wait of 20 seconds before the second, ...
- Delay (in seconds) before resetting the failure actions counter. The value -1 indicates that there will be no reset. For example, a value of 3600 means that after one hour the failure counter is reset and at the next failure, the 1st action in the list will be triggered.
- Message to be displayed on computers connected to it when the action "REBOOT" is triggered in case of service failure.
- "1" or "TRUE" to indicate that the service will be launched in delayed mode. This attribute is only applicable if the service is set to be launched automatically with the system. The delayed mode is used to tell Windows to launch the service after all automatic services (not delayed). This feature is only available in Windows Vista and higher.

This method returns a table of String whose elements correspond respectively to those enumerated previously. If this method is not defined, all this information will be empty, launching will be automatic and the stop will not be authorized. This method can be called several times by *JavaExe*.

7. **serviceControl_Pause** : This method is called when Windows tries to put in pause the service. This one will be indeed pauses about it if the method returns TRUE before a 30 seconds deadline. If the method is not declared, the service will be put in pause immediately.
8. **serviceControl_Continue** : This method is called when Windows tries to start again the service put in pause. This one will be indeed active if the method returns TRUE before a 30 seconds deadline. If the method is not declared, the service will be started again immediately.
9. **serviceControl_Stop** : This method is called when Windows tries to stop the service. This one will be stopped if the method returns TRUE before a 30 seconds deadline. After the stop of the service, the method **serviceFinish** will be finally called. If the method is not declared, the service will be stopped immediately.
10. **serviceControl_Shutdown** : This method is called when Windows is stopped or started again. It has the same behavior as **serviceControl_Stop**.

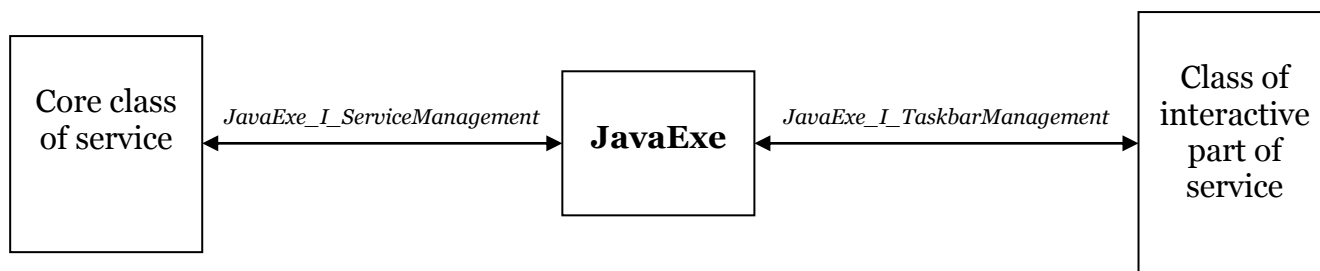
The 3 following methods are used for the services which interact with the Desktop. So that a service is recognized like interactive, it is enough that your Java application integrates the management of the taskbar (see the chapter "Management of the taskbar"). The service cannot communicate directly with the Desktop, it will have to pass by methods envisaged for this purpose :

11. **serviceDataFromUI** (*Serializable* data) : This method will be called by the interactive part of the service with in argument an object being treated by the service.
12. **serviceIsDataForUI** : This method will have to return TRUE if an object is available for the interactive part.
13. **serviceDataForUI** : This method returns an object for the interactive part.

With these three methods their counterpart in TaskbarManagement corresponds. See the chapter "Management of the taskbar" for the detail of these methods, as well as the *JavaExe_I_TaskbarManagement* interface in Appendix.

It is important to understand that there should not be direct bond between the classes of the service itself and the classes of its interactive part with the Desktop. If that were however to arrive, they will be two instances different from the same class and thus with different data.

Here a diagram summarizing the structure of an interactive service :



Of course, from the point of view of the Java developer all this is transparent. It will have simply to take care that its classes of the interactive part does not refer to the classes of the core part, and vice versa.

Running as Control Panel

So that your Java application is recognized like a Control panel, it is enough to create the .exe (see the chapter “General Use”) and to specify in the file properties, the property “RunType = 2”.

JavaExe is also provided with another type of file, **JavaExe.cpl**, which will have to be renamed as for the .exe. It is this file which will be recognized like a control panel by Windows.

It is possible to directly launch its installation, or its suppression, without passing by the possible dialog box of confirmation.

For that it is enough to launch **JavaExe.exe** (i.e. **MyApp.exe**) with like argument :

-installCPL : to force its installation
-deleteCPL : to force its suppression

Methods used as interface : JavaExe_I_ControlPanelManagement

These methods are directly called by **JavaExe** :

```
public static boolean cplIsCreate ();  
public static boolean cplIsDelete ();  
  
public static String[] cplGetInfo ();  
  
public static void cplOpen ();
```

These methods are to be declared either in the main class, or in a class with the same name but post fixed by “_ControlPanelManagement”. For example, if my main class is called *MyApp*, then these methods can be indifferently in *MyApp.class* or *MyApp_ControlPanelManagement.class*. It is not necessary to declare all them.

1. **cplIsCreate** : This method is called with launching of **JavaExe.exe** (i.e. **MyApp.exe**) if the control panel is not installed yet. It will be installed only if this method returns TRUE. If this method is not declared, a dialog box will open to require of the user if it or not wishes to install the control panel. The method **cplGetInfo** will be also called to obtain certain characteristics necessary to the creation of the control panel.
2. **cplIsDelete** : This method will be called with launching of **JavaExe.exe** (i.e. **MyApp.exe**) if the control panel is already installed. It will be removed only if this method returns TRUE. If this method is not declared, a dialog box will open to ask whether the user or not wishes to remove the control panel.
3. **cplGetInfo** : This method is called at the time of creation of the control panel in order to obtain certain additional information, such as :
 - Name.
 - Description.
 - Its categories of membership (starting from version XP of Windows). If you want to make appear your control panel in several categories, you will have to separate each value by a comma (`,`). See in Appendix the interface “**JavaExe_I_ControlPanelManagement**” for the list as of the these categories.

This method returns a table of String whose elements correspond respectively to those quoted previously. If this method is not defined, all this information will be empty and the name will be that of the .exe file.

4. **cplOpen** : This method will be called with the opening of the control panel. If this method is not declared, it is the *main* method which will be called.

Additional functionalities

System Events Management

This functionality of **JavaExe** makes it possible the Java application to receive some event of Windows, such as a connection or disconnection with a network, a change of display, a beginning or end of a session, ...

With this intention, it must exist a method which will be used as interface between **JavaExe** and your Java application, whose signature is form :

```
public static int notifyEvent (int msg, int val1, int val2, String val3, int[] arr1, byte[] arr2);
```

This method is to be declared either in the main class, or in a class of the same name but post fixed by “_SystemEventManager”. For example, if my main class is called *MyApp*, then this method can be indifferently in *MyApp.class* or *MyApp_SystemEventManager.class*.

The same method is used for all the types of events and its arguments depend on the received message. The value returned by **notifyEvent** also depends on the message.

The first argument, *msg*, contain the type of event and here is the list of the various values :

- **WM_COMPACTING** : This message is received when the system starts to saturate.
- **WM_CONSOLE** : This message is sent when *JavaExe* is launched in console mode (*JavaExe_console.exe*) and that an attempt at interruption with take place. The argument *val1* contains the type of interruption :
 - **CTRL_C_EVENT** : a CTRL-C is started, but will be cancelled if **notifyEvent** returns **0**.
 - **CTRL_BREAK_EVENT** : is used by Java for the dump active threads, but will be cancelled if **notifyEvent** returns **0**.
 - **CTRL_CLOSE_EVENT** : the user tries to close the DOS window and this attempt will be cancelled if **notifyEvent** returns **0**.
 - **CTRL_LOGOFF_EVENT** : the user with started the closing of its session. Some is the value returned by **notifyEvent** this closing will not be stopped.
 - **CTRL_SHUTDOWN_EVENT** : the user with started the shutdown of the system. Some is the value returned by **notifyEvent** the system will be stopped.
- **WM_DEVICECHANGE** : This message means that modification hardware occurred or requires a confirmation. For example if a peripheral were inserted or removed, or CD-Rom, ... The arguments used are :
 - *val1* : nature of the modification :
 - DBT_QUERYCHANGECONFIG**
 - DBT_CONFIGCHANGED**
 - DBT_CONFIGCHANGECANCELED**
 - DBT_DEVICEARRIVAL**
 - DBT_DEVICEQUERYREMOVE**
 - DBT_DEVICEQUERYREMOVEFAILED**
 - DBT_DEVICEREMOVECOMPLETE**
 - DBT_DEVICEREMOVEPENDING**

DBT_DEVICETYPESPECIFIC

- **val3** : name of port. Used only by **DBT_DEVTYP_PORT**.
- **arr1** : array of to the more 5 *int* (that depends on **val1**)
 - [0]**
 - [1]** = device type
DBT_DEVTYP_OEM
DBT_DEVTYP_VOLUME
DBT_DEVTYP_PORT
 - [2]**
 - [3]** = if **[1]=DBT_DEVTYP_VOLUME** => value where each binary position corresponds to a drive : bit 0 = drive A; bit 1 = drive B; ...; bit 26 = drive Z.
 - [4]** = if **[1]=DBT_DEVTYP_VOLUME** => **1**=media drive (CD-ROM, ...) ;
2=network drive
- **WM_DISPLAYCHANGE** : This event is received when the resolution of the screen changed. The arguments used are :
 - **val1** : the number of bits per pixel. One deducts from it the number of colors by 2^{val1} .
 - **val2** : a value on 32 bits broken up as follows: 31...16 15... 0. The bits from 0 to 15 correspond to the width of the screen. The bits from 16 to 31 give the height. To dissociate these 2 values, it is enough to apply :
 $w = (val2 \& 0x0000FFFF);$
 $h = ((val2 \gg 16) \& 0x0000FFFF);$
- **WM_ENDSESSION** : This message is received when the session of the user will be closed. Either because the user disconnects himself from his login, or that the system is stopped. This message is not received if the Java application is launched in console mode. The arguments used are :
 - **val1** : contains the value **1** if the closing of the session were confirmed, if not **0**. (see message **WM_QUERYENDSESSION** for this confirmation).
 - **val2** : allows to know if it acts of a simple disconnection of the user or stop of the system. If this argument contains value **ENDSESSION_LOGOFF** then it acts of a disconnection. It is preferable to test the presence of this value rather (as bits) than the strict equality :
 $((val2 \& \text{ENDSESSION_LOGOFF}) \neq 0)$ is preferable with
 $(val2 == \text{ENDSESSION_LOGOFF})$
- **WM_NETWORK** : This event is received when the state of the network changed. The arguments used are :
 - **val1** : the type of change
NET_DISCONNECT
NET_CONNECTING
NET_CONNECTED

- **val3** : the name of the interface network concerned.
- **arr1** : an array of 13 **int** used as follows :
 - [0]** = network type
 - MIB_IF_TYPE_OTHER**
 - MIB_IF_TYPE_ETHERNET**
 - MIB_IF_TYPE_TOKENRING**
 - MIB_IF_TYPE_FDDI**
 - MIB_IF_TYPE_PPP**
 - MIB_IF_TYPE_LOOPBACK**
 - MIB_IF_TYPE_SLIP**
 - [1...4]** = the 4 fields of client IP.
 - [5...8]** = the 4 fields of gateway IP.
 - [9...12]** = the 4 fields of network mask.
- **WM_POWERBROADCAST** : This event is started when the state of the battery or the power changed. The arguments used are :
 - **val1** : event type
 - PBT_APMQUERYSPEND**
 - PBT_APMQUERYSPENDFAILED**
 - PBT_APMSPEND**
 - PBT_APMRESUMECRITICAL**
 - PBT_APMRESUMESPEND**
 - PBT_APMBATTERYLOW**
 - PBT_APMPOWERSTATUSCHANGE**
 - PBT_APMOEMEVEN**
 - PBT_APMRESUMEAUTOMATIC**
 - **val2** : authorize or not an interaction with the user (as to display a dialog box...). If this argument contains **0** any interaction will be authorized.
 - **arr1** : contains 2 **int**
 - [0]** = a number of seconds currently usable out of battery.
 - [1]** = total of second usable one out of battery (maximum capacity of the battery).
 - **arr2** : contains 3 **byte**
 - [0]** = **1** if the battery is on the power A/C, if not **0**.
 - [1]** = state of charging of the battery (or 255 if unknown).
 - [2]** = percentage of charging (or 255 if unknown).
- **WM_QUERYENDSESSION** : This event is started when the session will be stopped. A confirmation is initially requested from the user and if **notifyEvent** returns **0** the session will not be stopped. Another message, **WM_ENDSESSION**, will be automatically sent in all the cases, after this one with the result of **notifyEvent**. This message is not received if the Java application is launched in console mode. The arguments used are :
 - **val2** : even significance that for message **WM_ENDSESSION**.

- **WM_SESSION_CHANGE** : This message is received when that a user connects himself, disconnects or locks the session. The arguments used are :
 - *val1* : the reason contains which started this event
 - WTS_SESSION_LOGGED**
 - WTS_CONSOLE_CONNECT**
 - WTS_CONSOLE_DISCONNECT**
 - WTS_REMOTE_CONNECT**
 - WTS_REMOTE_DISCONNECT**
 - WTS_SESSION_LOGON**
 - WTS_SESSION_LOGOFF**
 - WTS_SESSION_LOCK**
 - WTS_SESSION_UNLOCK**
 - WTS_SESSION_REMOTE_CONTROL**
 - *val2* : the number of the session contains concerned, if several sessions can be active at the same time.
 - *val3* : contains the name of the domain and the user (its login) who is at the origin of the event. This information is in format *domaine\login*.
 - *arr1* : is used only by **WTS_SESSION_LOGGED** and contains only one element indicating if the connected user is that which is currently active.
- **WM_SYSCOMMAND** : This event gathers various other events.
 - *val1* : the type of the event (for moment only **SC_SCREENSAVE**)
 - *val2* : state of screensaver : **1** for starting, **0** for stopping.
- **WM_TIMECHANGE** : This event takes place when the time of the system changed. No argument is used.
- **WM_USERCHANGED**

See in Appendices the *JavaExe_I_SystemEventManager* interface for the value of the constants used, and also examples 6 and 8 for events systems.

Taskbar Management

This functionality makes it possible the Java application to have his icon in the taskbar and possibly one or two menus associated (a menu for the right click and another for the left click).

Methods used as interface : JavaExe_I_TaskbarManagement

These methods are directly called by **JavaExe** :

```
public static String[][] taskGetMenu (boolean isRightClick, int menuID);
public static int taskGetDefaultMenuID (boolean isRightClick);

public static void taskDoAction (boolean isRightClick, int menuID);
public static boolean taskDisplayMenu (boolean isRightClick, Component parent, int x, int y);

public static String[] taskGetInfo ();
public static boolean taskIsShow ();
public static void taskInit ();

public static void taskDoBalloonAction ();
public static boolean taskIsBalloonShow ();
public static void taskSetBalloonSupported (boolean isSupported);
public static String[] taskGetBalloonInfo ();

public static void taskDataFromService (Serializable data);
public static boolean taskIsDataForService ();
public static Serializable taskDataForService ();
```

These methods are to be declared either in the main class, or in a class of the same name but post fixed by “_TaskbarManagement”. For example, if my main class is called *MyApp*, then these methods can be indifferently in *MyApp.class* or *MyApp_TaskbarManagement.class*.

It is not necessary to declare all them.

1. **taskGetMenu** (*boolean* isRightClick, *int* menuID) : This method is called to obtain the list of the entries of the menu associated with the icon in the taskbar. This menu will be managed by Windows, however if the Java application has and manages itself his menu for the icon, this method like **taskGetDefaultMenuID** and **taskDoAction** will be useless (cf the method **taskDisplayMenu** for the menus suitable for the application). In this case it will not be necessary to declare it, or then it will have to return the value `null`.

This method has 2 arguments :

- **isRightClick** : **TRUE** contains if the menu to be displayed corresponds to a right click of the mouse. There can thus be 2 different menus according to whether it acts of the right or left click.
- **menuID** : if the list of the entries to be returned corresponds to that of a sub-menu, this argument contains the number of the entry having this sub-menu. If not the value of the argument is to **0** (or negative).

The list returned by this method is of *String*[][] type, i.e. a list which contains a list of values of the *String* types. With each entry of the menu a list of values of the form corresponds:

{ ID, LABEL, TYPE, STATUS }

Where :

- *ID* = a single number for this entry. The value must be strictly higher than 0.
- *LABEL* = the text which will be displayed.
- *TYPE* = the nature of the entry (see lower).
- *STATUS* = the state of the entry to displaying (see lower).

With *TYPE* :

MFT_MENUBARBREAK	= place the entry in a new column with a vertical separation.
MFT_MENUBREAK	= place the entry in a new column without separation.
MFT_RADIOCHECK	= if the entry is in the state checked, it will then be displayed in the form of radio-button.
MFT_SEPARATOR	= display a horizontal separation. <i>LABEL</i> is then ignored.
MFT_RIGHTORDER	= display the text of the right-hand side towards the left.

If no value is specified, it will be then a simple text, aligned to left, which will be displayed. The values of the *TYPE* are mixables between them except **MFT_MENUBARBREAK** with **MFT_MENUBREAK**. For example :

{ *ID*, *LABEL*, ""+(**MFT_MENUBREAK** | **MFT_RIGHTORDER**), *STATUS* }

With *STATUS* :

MFS_DISABLED	= if the entry of the menu is disabled.
MFS_CHECKED	= if the entry is checked.
MFS_HILITE	= if the entry is preselected.

If no value is specified, the entry of the menu simply active and will then be unchecked. The values of *STATUS* are also mixables between them.

2. **taskGetDefaultMenuID** (*boolean* isRightClick) : This method makes it possible to define which is the entry of the menu which will be taken by default at the time of double-click on the icon. This entry will then be bold in the menu. If this method is not declared or if it returns 0 (or a negative value), no entry will be defined.

The argument of this method, *isRightClick*, contains **TRUE** if that relates to the menu for the right click of the mouse.

3. **taskDoAction** (*boolean* isRightClick, *int* menuID) : This method do the action to make when an entry of the menu will have been selected.

This method has 2 arguments :

- *isRightClick* : **TRUE** contains if the menu concerned is that of the right click of the mouse.
- *menuID* : the number of the entry selected by the user.

4. **taskDisplayMenu** (*boolean* isRightClick, *Component* parent, *int* x, *int* y) : This method make the displaying and the management of the menu associated with the icon. It returns **TRUE** if the menu is managed by this method.

This method has 4 arguments :

- *isRightClick* : **TRUE** contains if the menu to be managed corresponds to a right click of the mouse. There can thus be 2 different menus according to whether it acts of the right or left click.

- **parent** : according to the way in which the menu will be managed by the Java application, it can be necessary to have a relative object to which this menu will be attached. This relative object is created by **JavaExe**.
- **x** and **y** : co-ordinates where must be displayed the menu, corresponding to the corner lower right of this menu.

5. **taskGetInfo** : This method makes it possible to obtain various information for the displaying and the management of the icon and his menu. This method returns a table of *String* containing in the order :

- The description of the icon, which will be displayed when the mouse passes above.
- The type of action to be made for a simple right click of the mouse (by default it will be **ACT_CLICK_MENU**, see lower).
- The type of action to be made for a double right click of mouse (**ACT_CLICK_NOP** by default).
- The type of action to be made for a simple left click of mouse (**ACT_CLICK_NOP** by default).
- The type of action to be made for a double left click of mouse (**ACT_CLICK_OPEN** by default).

There are 3 types of possible action :

ACT_CLICK_NOP = do nothing
ACT_CLICK_OPEN = execute the action defined by the method **taskDoAction** with the entry of the menu returned by the method **taskGetDefaultMenuID**.
ACT_CLICK_MENU = display the menu by calling the method **taskDisplayMenu** initially. If the latter is not defined or returns **FALSE**, then the method **taskGetMenu** will be called.

6. **taskIsShow** : This method is regularly called by **JavaExe** to know if the icon must be displayed or hidden. If the method returns **TRUE** the icon will be displayed.
7. **taskInit** : This method is called at launching of the application.
8. **taskDoBalloonAction** : This method supports the action to take when a click has been held in the balloon.
9. **taskIsBalloonShow** : This method is regularly called by **JavaExe** to know if a message is ready to be displayed with the icon. If the method returns **TRUE**, then the method **taskGetBalloonInfo** will be called to obtain the message.
10. **taskSetBalloonSupported** (*boolean* isSupported) : This method is called at launching of the application to inform it if the version of Windows supports or not the balloon on icon. If the argument of this method contains **TRUE**, then the system supports the balloon management.
11. **taskGetBalloonInfo** : This method makes it possible to obtain the message of icon to be displayed and some information complementary. It will be called when the method **taskIsBalloonShow** returns **TRUE**, like at launching of the application. This method returns a table of *String* containing in the order :
- Title of message.
 - Message to display.
 - Type of message.
 - Duration of displaying of the message (in seconds).

With « Type of message » :

NIIF_NONE	= neutral message.
NIIF_INFO	= information message.
NIIF_WARNING	= warning message.
NIIF_ERROR	= error message.
NIIF_USER	= message with the application's icon.

12. **taskDataFromService** (*Serializable* data) : This method will be called by the service (if the application Java is launched in service mode) with in argument an object for the interactive part.
13. **taskIsDataForService** : This method will have to return **TRUE** if an object is available for the service part.
14. **taskDataForService** : This method returns an object for the service.

With these three methods their counterpart in `ServiceManagement` corresponds. See the chapter “Running as a Service” for the detail of these methods and some complementary explanations on the services in interaction with the Desktop, as well as the ***JavaExe_I_ServiceManagement*** interface in Appendix.

See in Appendices the ***JavaExe_I_TaskbarManagement*** interface for the value of the constants used, and also examples 4, 5 and 8 using the management of the taskbar.

Windows Registry Management

This feature allows the Java application to access the registry of Windows in both reading and writing and to perform all possible operations: creation and deletion of a key or value, read and modify a value, ...

Methods used as interface : JavaExe_I_RegistryManagement

To do this, simply declare some static methods in the main class or in a class with the same name but post fixed with "_RegistryManagement". For example, if my main class is called MyApp, then these methods may be declared either in MyApp.class or MyApp_RegistryManagement.class.

It is not necessary all declare them, only those that the Java application needs. It should be noted that these methods are native type, that is to say that it is not necessary to define the method bodies, but only their signature, such as:

```
public static native String nativeReg_GetValueSTR (int hkey, String pathKey, String nameValue, boolean isExpandVal);
public static native byte[] nativeReg_GetValueBIN (int hkey, String pathKey, String nameValue);
public static native int nativeReg_GetValueDWORD (int hkey, String pathKey, String nameValue);
public static native long nativeReg_GetValueQWORD (int hkey, String pathKey, String nameValue);
public static native String[] nativeReg_GetValueMULTI (int hkey, String pathKey, String nameValue);

public static native boolean nativeReg_SetValueSTR (int hkey, String pathKey, String nameValue, String val, boolean
isTypeExpand);
public static native boolean nativeReg_SetValueBIN (int hkey, String pathKey, String nameValue, byte[] val);
public static native boolean nativeReg_SetValueDWORD (int hkey, String pathKey, String nameValue, int val, boolean
isTypeBigEndian);
public static native boolean nativeReg_SetValueQWORD (int hkey, String pathKey, String nameValue, long val);
public static native boolean nativeReg_SetValueMULTI (int hkey, String pathKey, String nameValue, String[] val);

public static native int nativeReg_GetTypeValue (int hkey, String pathKey, String nameValue);

public static native boolean nativeReg_CreateKey (int hkey, String pathKey);
public static native boolean nativeReg_DeleteKey (int hkey, String pathKey);
public static native boolean nativeReg_DeleteValue (int hkey, String pathKey, String nameValue);

public static native String[] nativeReg_EnumKeys (int hkey, String pathKey);
public static native String[] nativeReg_EnumValues (int hkey, String pathKey);
```

In general, the arguments **hkey** correspond to the root from which starts the key, possible values are found to the Appendix in the *JavaExe_I_RegistryManagement* interface. The commonly used values are *HKEY_CURRENT_USER* and *HKEY_LOCAL_MACHINE*. For further explanation of these values, visit the Microsoft MSDN®: <http://msdn.microsoft.com/fr-fr/library/ms724836.aspx>.

Then the arguments **pathKey** correspond to a path to access a key (excluding the value name). The names of keys are separated by a back-slash ('\'). For example: "Software\JavaExe\Examples".

The arguments **nameValue** correspond to a value name.

And finally the arguments **val** is the value that you want to associate to **nameValue**. Its type depends on the method used. See <http://msdn.microsoft.com/fr-fr/library/ms724884.aspx> for more information on the types in the Windows Registry.

Also note that numbers can be stored in the registry in two formats. For example a 32-bit integer (0x12345678) will be stored:

- « Little Endian » : in the form 0x78 0x56 0x34 0x12
- « Big Endian » : in the form 0x12 0x34 0x56 0x78

Native methods to be declared:

Methods to retrieve values:

1. **nativeReg_GetValueSTR**: retrieves a *String* value associated to **nameValue** located at the end of **pathKey** path. The value stored in the Windows registry must be of type REG_SZ, REG_EXPAND_SZ or REG_LINK, otherwise the method returns **null**. The argument **isExpandVal** is used in the case of a REG_EXPAND_SZ value and can interpret (or not) the environment variables contained in the value. If the value is not found from the specified path, the method returns **null**.
2. **nativeReg_GetValueBIN**: retrieves the value as a *byte* array corresponding to the data stored as is without interpretation or processing. The type of the value stored in the Windows Registry does not matter. If the value is not found the method returns **null**.
3. **nativeReg_GetValueDWORD**: to retrieve the value to *int* (32 bits). The value stored in the Windows Registry must be of type REG_DWORD or REG_DWORD_BIG_ENDIAN. For the latter, the value is automatically converted to "Little Endian" which is the standard format of numbers in Windows. If the value is of another type or does not exist, the method will return 0.
4. **nativeReg_GetValueQWORD**: retrieves a value to type *long* (64 bits). The value stored in the Windows Registry must be a REG_QWORD. If the value is of another type or does not exist, the method will return 0.
5. **nativeReg_GetValueMULTI**: retrieves a value in a *String* array. The value stored in the Windows Registry must be of type REG_MULTI_SZ. If the value is of another type or does not exist, the method returns **null**.

Methods to change the values:

6. **nativeReg_SetValueSTR**: modify or create a value of type REG_SZ, or REG_EXPAND_SZ if the argument **isTypeExpand** is true. If the path **pathKey** contains keys that do not exist, these will be created. The method returns **true** if the operation was successful.
7. **nativeReg_SetValueBIN**: modify or create a value of type REG_BINARY. The keys path will be created if necessary. The method returns **true** if the operation was successful.
8. **nativeReg_SetValueDWORD**: modify or create a value of type REG_DWORD, or REG_DWORD_BIG_ENDIAN if the argument **isTypeBigEndian** is **true**. The keys path will be created if necessary. The method returns **true** if the operation was successful.
9. **nativeReg_SetValueQWORD**: modify or create a value of type REG_QWORD (64 bits). The keys path will be created if necessary. The method returns **true** if the operation was successful.
10. **nativeReg_SetValueMULTI**: modify or create a value of type REG_MULTI_SZ. The keys path will be created if necessary. The method returns **true** if the operation was successful.

Method to retrieve information about the values:

11. **nativeReg_GetTypeValue**: provides the type of a value stored in the Windows Registry. The return types are:

REG_NONE (= 0) : if the value is not found from the specified path.
 REG_SZ (= 1) : *String (Unicode)*.
 REG_EXPAND_SZ (= 2) : *String (Unicode) containing environment variables*.
 REG_BINARY (= 3) : *raw binary data*.
 REG_DWORD (= 4) : *32-bits integer*.
 REG_DWORD_BIG_ENDIAN (= 5) : *32-bits integer in format « BigEndian »*.
 REG_LINK (= 6) : *String (Unicode) corresponding to a symbolic link to Registry (should not be used)*.
 REG_MULTI_SZ (= 7) : *list of Strings (Unicode)*.
 REG_QWORD (= 11) : *64-bits integer*.

Visit the Microsoft® MSDN <http://msdn.microsoft.com/fr-fr/library/ms724884.aspx> for more information on the types in the Windows Registry.

Methods for the creation or deletion:

12. **nativeReg_CreateKey**: create nonexistent keys in the specified path **pathKey**. The method returns **true** if the operation was successful or if all the keys already exist.
13. **nativeReg_DeleteKey**: Deletes a key located at the end of the path **pathKey**. This key can hold values, which will be removed with it, but must not have subkeys. The method returns **true** if the operation was successful.
14. **nativeReg_DeleteValue**: Deletes the value named **nameValue** in Windows Registry. The method returns **true** if the operation was successful.

Methods to retrieve the list of names:

15. **nativeReg_EnumKeys**: returns the list of subkeys contained directly in the path **pathKey**. The method will return **null** if the specified path is invalid.
16. **nativeReg_EnumValues**: returns a list of value names contained directly in the path **pathKey**. The method will return **null** if the specified path is invalid.

See in Appendices, the **JavaExe_I_RegistryManagement** interface for the value of the constants used, and also the examples 9 and 10 which using the Windows Registry management.

Appendices

Java Interfaces

These interfaces are used only to have access to the constants necessary in the various functionalities of **JavaExe**. The classes methods which are defined here are there only as an indication since the methods defined in an interface apply only to instances.

ApplicationManagement

```
interface JavaExe_I_ApplicationManagement
{
    public static boolean isCloseSplash ();
    public static boolean isOneInstance (String[] args);

    public static boolean sessionIsRestore();
    public static String[] sessionGetMainArgs();
    public static Serializable sessionGetData();
    public static void sessionSetData (Serializable data);
}
```

ControlPanelManagement

```
interface JavaExe_I_ControlPanelManagement
{
    static final int CATGR_NONE           = -1;
    static final int CATGR_OTHER          = 0;
    static final int CATGR_THEMES         = 1;
    static final int CATGR_HARDWARE       = 2;
    static final int CATGR_NETWORK        = 3;
    static final int CATGR_SOUND          = 4;
    static final int CATGR_PERF           = 5;
    static final int CATGR_REGIONAL       = 6;
    static final int CATGR_ACCESS         = 7;
    static final int CATGR_PROG           = 8;
    static final int CATGR_USER           = 9;
    static final int CATGR_SECURITY       = 10;

    public static boolean cplIsCreate ();
    public static boolean cplIsDelete ();

    public static String[] cplGetInfo ();

    public static void cplOpen ();
}
```

RegistryManagement

interface **JavaExe_I_RegistryManagement**

```
{
    static final int HKEY_CLASSES_ROOT          = 0x80000000;
    static final int HKEY_CURRENT_USER          = 0x80000001;
    static final int HKEY_LOCAL_MACHINE         = 0x80000002;
    static final int HKEY_USERS                 = 0x80000003;
    static final int HKEY_PERFORMANCE_DATA     = 0x80000004;
    static final int HKEY_CURRENT_CONFIG       = 0x80000005;
    static final int HKEY_DYN_DATA             = 0x80000006;

    static final int REG_NONE                   = 0;
    static final int REG_SZ                     = 1;
    static final int REG_EXPAND_SZ             = 2;
    static final int REG_BINARY                = 3;
    static final int REG_DWORD                 = 4;
    static final int REG_DWORD_BIG_ENDIAN     = 5;
    static final int REG_LINK                  = 6;
    static final int REG_MULTI_SZ             = 7;
    static final int REG_QWORD                 = 11;

    public static native String nativeReg_GetValueSTR (int hkey, String pathKey, String nameValue
        ,boolean isExpandVal);
    public static native byte[] nativeReg_GetValueBIN (int hkey, String pathKey, String nameValue);
    public static native int nativeReg_GetValueDWORD (int hkey, String pathKey, String nameValue);
    public static native long nativeReg_GetValueQWORD (int hkey, String pathKey, String nameValue);
    public static native String[] nativeReg_GetValueMULTI (int hkey, String pathKey, String nameValue);

    public static native boolean nativeReg_SetValueSTR (int hkey, String pathKey, String nameValue, String val
        ,boolean isTypeExpand);
    public static native boolean nativeReg_SetValueBIN (int hkey, String pathKey, String nameValue, byte[] val);
    public static native boolean nativeReg_SetValueDWORD (int hkey, String pathKey, String nameValue, int val
        ,boolean isTypeBigEndian);
    public static native boolean nativeReg_SetValueQWORD (int hkey, String pathKey, String nameValue
        ,long val);
    public static native boolean nativeReg_SetValueMULTI (int hkey, String pathKey, String nameValue
        ,String[] val);

    public static native int nativeReg_GetTypeValue (int hkey, String pathKey, String nameValue);

    public static native boolean nativeReg_CreateKey (int hkey, String pathKey);
    public static native boolean nativeReg_DeleteKey (int hkey, String pathKey);
    public static native boolean nativeReg_DeleteValue (int hkey, String pathKey, String nameValue);

    public static native String[] nativeReg_EnumKeys (int hkey, String pathKey);
    public static native String[] nativeReg_EnumValues (int hkey, String pathKey);
}
```

ServiceManagement

interface **JavaExe_I_ServiceManagement**

```
{
    public static boolean serviceIsCreate ();
    public static boolean serviceIsLaunch ();
    public static boolean serviceIsDelete ();

    public static boolean serviceControl_Pause ();
}
```

```

public static boolean serviceControl_Continue ();
public static boolean serviceControl_Stop ();
public static boolean serviceControl_Shutdown ();

public static String[] serviceGetInfo ();
public static boolean serviceInit ();
public static void serviceFinish ();

public static void serviceDataFromUI (Serializable data);
public static boolean serviceIsDataForUI ();
public static Serializable serviceDataForUI ();
}

```

SystemEventManager

```

interface JavaExe_I_SystemEventManager
{
    static final int WM_QUERYENDSESSION          = 0x0011;
    static final int WM_ENDSESSION                = 0x0016;
    static final int WM_DEVMODECHANGE            = 0x001B;
    static final int WM_TIMECHANGE               = 0x001E;
    static final int WM_COMPACTING               = 0x0041;
    static final int WM_USERCHANGED              = 0x0054;
    static final int WM_DISPLAYCHANGE            = 0x007E;
    static final int WM_SYSCOMMAND               = 0x0112;
    static final int WM_POWERBROADCAST           = 0x0218;
    static final int WM_DEVICECHANGE             = 0x0219;
    static final int WM_SESSION_CHANGE           = 0x02B1;
    static final int WM_NETWORK                  = 0x0401;
    static final int WM_CONSOLE                  = 0x0402;

    static final int PBT_APMQUERYSUSPEND          = 0x0000;
    static final int PBT_APMQUERYSUSPENDFAILED    = 0x0002;
    static final int PBT_APMRESUMECRITICAL        = 0x0006;
    static final int PBT_APMRESUMESUSPEND        = 0x0007;
    static final int PBT_APMBATTERYLOW            = 0x0009;
    static final int PBT_APMPOWERSTATUSCHANGE     = 0x000A;
    static final int PBT_APMOEMEVENT             = 0x000B;
    static final int PBT_APMRESUMEAUTOMATIC       = 0x0012;

    static final int DBT_QUERYCHANGECONFIG        = 0x0017;
    static final int DBT_CONFIGCHANGED            = 0x0018;
    static final int DBT_CONFIGCHANGECANCELED    = 0x0019;
    static final int DBT_DEVICEARRIVAL           = 0x8000;
    static final int DBT_DEVICEQUERYREMOVE       = 0x8001;
    static final int DBT_DEVICEQUERYREMOVEFAILED = 0x8002;
    static final int DBT_DEVICEREMOVECOMPLETE    = 0x8004;
    static final int DBT_DEVICEREMOVEPENDING     = 0x8003;
    static final int DBT_DEVICEYPESPECIFIC       = 0x8005;

    static final int DBT_DEVTYP_OEM               = 0x00000000;
    static final int DBT_DEVTYP_VOLUME           = 0x00000002;
    static final int DBT_DEVTYP_PORT             = 0x00000003;

    static final int ENDSESSION_LOGOFF           = 0x80000000;

    static final int SC_SCREENSAVE                = 0xF140;

    static final int NET_DISCONNECT              = 0;
    static final int NET_CONNECTING              = 1;
    static final int NET_CONNECTED               = 2;

```

```

static final int MIB_IF_TYPE_OTHER          = 1;
static final int MIB_IF_TYPE_ETHERNET      = 6;
static final int MIB_IF_TYPE_TOKENRING     = 9;
static final int MIB_IF_TYPE_FDDI          = 15;
static final int MIB_IF_TYPE_PPP           = 23;
static final int MIB_IF_TYPE_LOOPBACK      = 24;
static final int MIB_IF_TYPE_SLIP          = 28;

static final int WTS_SESSION_LOGGED         = 0;
static final int WTS_CONSOLE_CONNECT       = 1;
static final int WTS_CONSOLE_DISCONNECT    = 2;
static final int WTS_REMOTE_CONNECT        = 3;
static final int WTS_REMOTE_DISCONNECT     = 4;
static final int WTS_SESSION_LOGON         = 5;
static final int WTS_SESSION_LOGOFF        = 6;
static final int WTS_SESSION_LOCK          = 7;
static final int WTS_SESSION_UNLOCK        = 8;
static final int WTS_SESSION_REMOTE_CONTROL = 9;

static final int CTRL_C_EVENT               = 0;
static final int CTRL_BREAK_EVENT          = 1;
static final int CTRL_CLOSE_EVENT          = 2;
static final int CTRL_LOGOFF_EVENT         = 5;
static final int CTRL_SHUTDOWN_EVENT       = 6;

public static int notifyEvent (int msg, int val1, int val2, String val3, int[] arr1, byte[] arr2);
}

```

TaskbarManagement

```

interface JavaExe_I_TaskbarManagement
{
    static final int ACT_CLICK_NOP          = 0;
    static final int ACT_CLICK_OPEN         = 1;
    static final int ACT_CLICK_MENU         = 2;

    static final int NIIF_NONE              = 0;
    static final int NIIF_INFO              = 1;
    static final int NIIF_WARNING          = 2;
    static final int NIIF_ERROR            = 3;
    static final int NIIF_USER             = 4;

    static final int MFT_MENUBARBREAK       = 0x0020;
    static final int MFT_MENUBREAK         = 0x0040;
    static final int MFT_RADIOCHECK         = 0x0200;
    static final int MFT_SEPARATOR         = 0x0800;
    static final int MFT_RIGHTORDER        = 0x2000;

    static final int MFS_DISABLED           = 0x0003;
    static final int MFS_CHECKED           = 0x0008;
    static final int MFS_HILITE            = 0x0080;

    public static String[][] taskGetMenu (boolean isRightClick, int menuID);
    public static int taskGetDefaultMenuID (boolean isRightClick);

    public static void taskDoAction (boolean isRightClick, int menuID);
    public static boolean taskDisplayMenu (boolean isRightClick, Component parent, int x, int y);

    public static String[] taskGetInfo ();
}

```

```
public static boolean taskIsShow ();  
public static void taskInit ();  
  
public static void taskDoBalloonAction ();  
public static boolean taskIsBalloonShow ();  
public static void taskSetBalloonSupported (boolean isSupported);  
public static String[] taskGetBalloonInfo ();  
  
public static void taskDataFromService (Serializable data);  
public static boolean taskIsDataForService ();  
public static Serializable taskDataForService ();  
}
```


Examples

These examples are there only to show in practice the various functionalities of **JavaExe**. Their source code can be used as starting point for more complex applications.

1 - Application

This example shows simply a Java application which opens a dialog box. It introduces the notion of the .EXE to launch a Java application with a splash screen.

2 - Control Panel

This example define a control panel Windows containing a dialog box has 3 tabs, where their values will be read since the properties file associated to the application (*Example2.properties*). While pressing on the button “Apply” or “Ok” the values will be saved in this same file.

There are 2 ways to launch this control panel. Either in double clicking on the .EXE for install / uninstall the control panel, or in double clicking on the .CPL to open it directly.

3 - Service

This example define a service without Desktop interaction. At the time of its installation, a dialog box proposes various parameters for the launching of the service, like the number of port. This service standby of connection on the port thus defined and return the date to the client who connected.

4 - TrayIcon

This example shows a Java application using the taskbar management. The icon on the taskbar has 2 menus, for the right click and for the left click of the mouse. In double clicking on the icon, the application opens a dialog box in which a checkbox is to display or not the icon in the taskbar.

5 - Service & TrayIcon

This example define a service with Desktop interaction. This service is the same one as for example 3. The interaction with the Desktop occurs by the taskbar whose icon will have a menu on the right click of the mouse. Since this menu it will be possible to reconfigure the service or launching a browser on the port of listening service.

6 - System Event

This example, which launches like a simple application, intercepts the events systems of Windows and display them in a dialog box. If the application is launched in console Dos, with *Example6_console.exe*, the CTRL-C of the console will be also intercepted and a dialog box will open to require the confirmation of it.

7 - OneInstance

This example shows the functionality of the “OneInstance” making it possible to control the number of instances of the application launched at the same time. When the application launches, a dialog box opens containing a checkbox. When this one is checked several instances of the application will be able to be executed at the same time.

If the application is launched with arguments, those will be displayed initially in dialog box of the 1st instance, then according to whether the checkbox is checked or not these same arguments are displayed in dialog box of the application newly launched.

8 - Service & TrayIcon & System Event

This example creates a Windows service with Desktop interaction intercepting the events systems. It thus manages the taskbar whose icon has a menu on the right click of the mouse. From this menu it is possible to open a dialog box displaying the events received by the service or to hide the messages related to this icon.

9 - Registry

This example put into practice the management of Windows registry. In the window that opens, a tree structure is built from the HKEY_CURRENT_USER. It will be possible to create keys or values (only type REG_SZ), automatically prefixed with "JavaExe -", or remove only those that have this same prefix (to avoid mishandling of the Registry).

10 - Test Unicode

This example verifies that **JavaExe** handles Unicode, whether in the arguments received by the application, or in the Windows Registry, or in the Java system properties, or in the filename. EXE or . JAR.

11 - Restore Session

This example illustrates the restoring of the Java application after a system restart. When the application opens, just type some text and to reboot Windows without exiting the Java application. After rebooting the system, the application will automatically be restarted with the text that has been entered before the restart.

JavaExe and **UpdateRsrcJavaExe** (ex-**MergeICO**) are creations and copyrighted of DevWizard (DevWizard@free.fr).

You are free to be useful to you about it and to provide them with your applications which you wish to distribute, who they are freeware, shareware or commercial. Only DevWizard is authorized to make an unspecified modification to **JavaExe** and **UpdateRsrcJavaExe**.

Any request for modification can be made by mail to DevWizard@free.fr

© 2002-2012 by DevWizard