JavaExe

version 3.2

by DevWizard
(DevWizard@free.fr)

(12 October 2013)

to Jawaher...

Table of Contents

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, or as a ScreenSaver.

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***, addition to the different types of launch, we can note :

- limitation 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, ...),
- managenent of the taskbar (this allows to have a interactive service),
- management of the Windows registry,
- opportunity to restart the Java application (or part of it) in Administrator mode,
- access to certain system functions to restart Windows, or to put it in standby, ...
- ability to control Windows services,
- dynamic splash screen

In this documentation, the icon represents a new feature or a change from the previous version of ***JavaExe***.

« Imagination is more important than Knowledge »
Albert Einstein

History

- **3.2** *(12 October 2013)* :
  - Support for 64-bit JRE.
  - Screensaver : ability to run the Java application as a screensaver (with **JavaExe.scr** file)
  - Ability to fully restart the Java application in Admin mode, controlled in code or through the manifest of the executable.
  - Ability to run only a part of the Java application in Admin mode.
  - Management of Services Control: ability to manage Windows services, delete, stop, retrieve status information, change their configuration, ...
  - Shutdown Management of the PC, or the standby, locking the session, ...
  - Possibility to prevent running of the screensaver, turns off the monitor, and the automatic standby of the PC.
  - Dynamic splash screen: can update in real time the static splash screen.
  - Static splash screen: additional image formats such as GIF, JPG and PNG.
  - Taskbar:
    - Addition of MFS_ADMIN attribute for menu entries to have the elevation icon request for Admin mode.
    - The ***taskInit*** method now has an argument to know it is in ServiceUI mode or not.
  - Automatic detection of the property "*RunType*" according to the features used.
  - Service :
    - Adding a control argument, -startServiceUI, to launch manually the UI part of the interactive service.
    - Automatic detection of configuration changes since the creation of the service.
    - Fixed a major bug for interactive services: if the ***taskGetInfo*** method was not declared, the UI part of the service does not launch.
  - System Events :
    - SC_MONITORPOWER: new event for the monitor turns off.
    - PBT_APMPOWERSTATUSCHANGE: was not intercepted on Windows Vista and higher.
    - SC_SCREENSAVE: was not intercepted from a service.
  - Adding arguments **JavaExe.exe** (and its derivatives) :
    - -createShortcut : creating shortcuts to the executable file, depending on the features used
    - -deleteShortcut : delete shortcuts created with -createShortcut
  - **UpdateRsrcJavaExe** :
    - Integrating files for **JavaExe.scr**
    - Integration of a manifest in the executable for the Admin mode
    - Adding arguments : -admin ; -scr ; -img (replace -bmp)
    - If moving files to integrate is on **JavaExe.exe** (or derivatives), then **UpdateRsrcJavaExe** will automatically start to integrate these files.
    - Fixed a minor bug on the arguments received between " or '
    - Fixed a minor bug on Drag'n'Drop files on **UpdateRsrcJavaExe** (on some versions of Windows and under certain conditions of starting).
  - Fixed a major bug with the feature ***IsOneInstance*** when the first instance of the application was launched in administrator mode but not the other instances.
  - Fixed a minor bug when uninstalling an interactive service: the icon on the taskbar remained active sometimes (on some versions of Windows).
  - Fixed a minor bug when **JavaExe** was not found a JRE: the browser does not open for downloading and install a JRE (concerned only some versions of Windows).

- **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)*: birth of **JavaExe**.

License

The term **JavaExe** includes the executable files **JavaExe.exe**, **JavaExe_console.exe**, **JavaExe.cpl**, **JavaExe.scr** (and their derivative, that is to say, their renamed version by the name of the main class or .jar) and the file **UpdateRsrcJavaExe.exe**.

License of use

JavaExe is freeware and as such you are allowed to use personal way, or in a professional or educational background (university, school, ...).

Redistribution license

You are also allowed to redistribute **JavaExe** with your Java application, be it commercial or freeware.

Modifications allowed and not allowed

The only modifications allowed are those specified in this documentation, or adding resources such as `RT_MANIFEST`, `RT_VERSION`, ...
It is also authorized to add a digital signature to executable files of **JavaExe** or its derivatives (except **UpdateRsrcJavaExe.exe**) using utilities provided for this purpose.

However, you are not allowed to modify the binary code (that is to say, the executable part) of files **JavaExe**.

General Use

Creation of .EXE

Above all, it is important to note that there are two versions of a ***JavaExe*** executable: a 32-bit version (or x86) and another 64-bit (or x64). The 32-bit version is planned for Windows 32-bit and also works on a 64-bit system, but with a 32-bit JRE in both cases. The other version, the 64-bit, works only on 64-bit systems and only with 64-bit JRE.

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 uncompacted 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.

<u>*Example :*</u>

> 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.
  <u>*example*</u> *:*
  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
  3 = as a SreenSaver
  <u>*example*</u> *:*
  RunType = 1 *JavaExe will launch the application as Service*

  However the *RunType* can be determined automatically depending on the features of JavaExe used in the Java application:

  - if **serviceGetInfo()** or **serviceInit()** is declared, then **RunType = 1**
  - if the **JavaExe.cpl** file is used, then **RunType = 2**
  - if the **JavaExe.scr** file is used and **scrsvPaint()** is declared, then **RunType = 3**

- **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).*

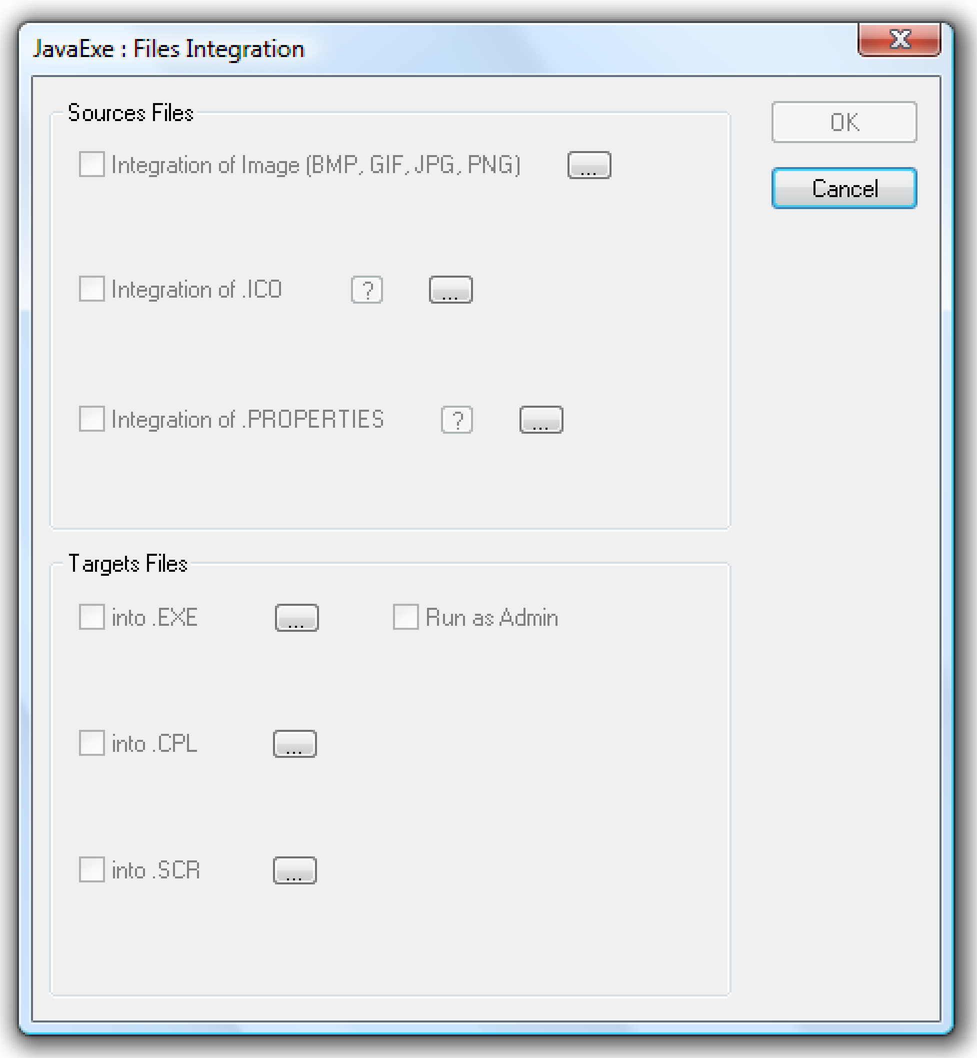
- **Display BoxInstall (**or **Display_BoxInstall)** = *to indicate if a message must be displayed when JavaExe does not find a JRE or JDK, and asking whether one wishes to install a JRE or to leave the application. Only two values are accepted : 0 or 1.*
  1 = display dialog box to install or not the 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, in the .cpl (for the use of the Java application as Control Panel), or in the .scr (for ScreenSaver).


JavaExe : Files Integration
Sources Files
Integration of Image (BMP, GIF, JPG, PNG)
Integration of .ICO
Integration of .PROPERTIES
Targets Files
into .EXE
Run as Admin
into .CPL
into .SCR
OK
Cancel

The integration of these files can be done in four 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.
- By moving the desired files on **JavaExe.exe** (or the renamed version **MyApp.exe**), provided that **UpdateRsrcJavaExe** is present and the same folder that **JavaExe.exe** (or **MyApp.exe**).

Types of recognized files :

- **.BMP** ; **.GIF** ; **.JPG** ; **.PNG** : *allows to define a splash screen in the Java application.*

- **.ICO** : *allows to change the icon of the .exe file, or .scr 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).*

- **.SCR** : *allows to specify the .scr derived from JavaExe.scr (renamed or not) which will receive the files to be integrated (only the files of icon and properties can be integrated into a .scr).*

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 OK 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.*

- **-scr=***file* : *to indicate the name of a .scr file which will receive the files to be integrated. This file must be a derived of JavaExe.scr.*

- **-ico=***file* : *allows to indicate the name of an icon which will be integrated into the .exe or .scr*

- **-img=***file* : *allows to indicate the name of an image, in format BMP, GIF, JPG or PNG, which will be integrated into the .exe and being used as splash screen.*

- **-bmp=***file* : *same feature as -img, but should not be used.*

- **-prp=***file* : *to specify the name of a properties file which will be integrated.*

- **-admin**=*true* (or *1*) : *includes a manifest to run the executable in Admin mode.*
- **-admin**=*false* (or *0*) : *includes a manifest to run the executable in standard mode.*
- **-admin** : *like as «* **-admin**=*true ».*

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**.

To do this simply use **UpdateRsrcJavaExe**, provided with **JavaExe**, either by command line with arguments **-ico=***icon file* and **-exe=***executable file*, either by moving the icon and executable files on the window of **UpdateRsrcJavaExe** (see the previous paragraph for use).

Splash Screen

To define a splash screen in your Java application it is enough to have the image in format BMP, GIF, JPG or PNG and to use the **UpdateRsrcJavaExe** program, either by command line with arguments **-img=***image file* and **-exe=***executable file*, either by moving the image and executable files on the window of **UpdateRsrcJavaExe** (see the paragraph of this utility).

This defined splash screen will be static, that is to say that the same screen will be displayed for a while. However, while this screen is displayed, it is possible to change it at regular time intervals to give it a dynamic appearance or to automatically associate a progress bar (see Section "Dynamic Splash Screen" page 45).

Creating shortcuts

It is possible to automatically create shortcuts on **JavaExe** executable files depending on the features used by the Java application.

To do this, simply call in command line the **JavaExe** derivative (that is to say, its renamed version in **MyApp.exe**) and to pass the following arguments :

- **-createShortcut** : to create the necessary shortcuts according using :
    - Service :
      *-install.lnk et *-delete.lnk : to install and remove the service.
      *-start.lnk : to start the service, if STOP is allowed or if it is not automatic.
      *-stop.lnk : to stop the service if the STOP is authorized.
      *-runUI.lnk : to launch the UI part of the service, if it is interactive.
    - Control Panel :
      *-install.lnk et *-delete.lnk : to install it or remove it.
    - ScreenSaver :
      *-install.lnk et *-delete.lnk : to install and remove the screen saver.
      *-config.lnk : to open the setup screen, if exists.

- **-deleteShortcut** : to delete shortcuts created with the command **-createShortcut**.

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.

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 lauching 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
-startServiceUI	: to run its UI part if the service is interactive

Methods used as interface : *JavaExe_I_ServiceManagement*

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

1. public static *boolean* **serviceIsCreate** ();
2. public static *boolean* **serviceIsLaunch** ();
3. public static *boolean* **serviceIsDelete** ();

4. public static *boolean* **serviceInit** ();
5. public static *void* **serviceFinish** ();

6. public static *String[]* **serviceGetInfo** ();

7. public static *boolean* **serviceControl_Pause** ();
8. public static *boolean* **serviceControl_Continue** ();
9. public static *boolean* **serviceControl_Stop** ();
10. public static *boolean* **serviceControl_Shutdown** ();

11. public static *void* **serviceDataFromUI** (*Serializable* data);
12. public static *boolean* **serviceIsDataForUI** ();
13. 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***.

This method will be called every 30 seconds during the execution of the service to automatically detect any configuration changes compared with the value returned by the same method when creating the service. If a change is detected, the new configuration will be applied without the service is stopped.

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**.

Interactive Service

A Windows service cannot be directly interactive with the Desktop from at least one Windows Vista for security reasons (it was still possible to Windows XP).

The solution is to separate the actual services part of its interactive part (windows, dialog box, user interaction, ...). The first part will always run as a service, while the second will be launched as a Windows application in the context of the current user. This will involve two different processes that must communicate between.

JavaExe automatically manages these two parts and their communication to exchange data or actions to perform. For example, since the service cannot display itself an error message, it signal to the interactive part that it must display the error message. And, the interactive part can send requests for actions to be performed by the service according to user choice.

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.

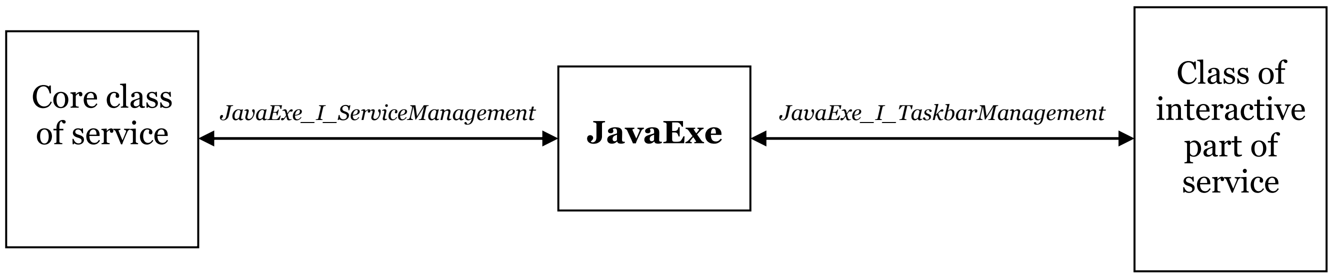
12. **serviceIsDataForUI** : This method will have to return TRUE if an object is available for the interactive part.

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 :


Core class of service
JavaExe_I_ServiceManagement
JavaExe
JavaExe_I_TaskbarManagement
Class of interactive part of 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.

Running as ScreenSaver

For that your Java application is recognized like a screensaver, just create the .exe (See the chapter "General Use") and specify in the file .properties, the property "RunType = 3".
JavaExe also comes with another file type, the **JavaExe.scr**, which will be renamed like for the .exe. This is the file that will be recognized as a screensaver for Windows.

You can start the installation directly, or its removal, bypassing any confirmation dialog boxes. To do this simply run ***JavaExe.exe*** (that is to say ***MyApp.exe***) with the argument :

-installSCR	: to force its installation
-deleteSCR	: to force its suppression
-configSCR	: to force open the setup screen, if existing

Methods used as interface : *JavaExe_I_ScreenSaverManagement*

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

public static *boolean* **scrsvIsCreate** ();
public static *boolean* **scrsvIsDelete** ();

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

public static *void* **scrsvInit** ();
public static *void* **scrsvFinish** ();

public static *void* **scrsvOpenConfig** ();
public static *void* **scrsvPaint** (*Graphics2D* g, *int* wScr, *int* hScr);

public static *boolean* **scrsvIsExitByKey** (*int* keycode, *boolean* isUp);
public static *boolean* **scrsvIsExitByMouse** (*int* x, *int* y, *int* nbClick, *int* button, *boolean* isUp);

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

1. **scrsvIsCreate** : This method is called with launching of ***JavaExe.exe*** (that is to say ***MyApp.exe***) if the screensaver 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 asking the user whether or not to install the screensaver.
   The method **scrsvGetInfo** will also be called to get some informations to create the screensaver.

2. **scrsvIsDelete** : This method will be called with launching of ***JavaExe.exe*** (that is to say ***MyApp.exe***) if the screensaver is already installed. It will be removed only if this method returns `TRUE`. If this method is not declared, a dialog box will open asking if the user wants or not to remove the screensaver.

3. **scrsvGetInfo** : This method is called when creating the screensaver to get some additional information, such as :
   - Description.

- Transparency : « **1** » or « **TRUE** » to indicate that the screensaver will have a transparent background, otherwise a black background will be put by default.
- Mouse : « **1** » or « **TRUE** » to indicate that the mouse pointer should be visible.
- Frequency refresh : millisecond value corresponding to the refresh rate, that is to say the wait time between calls to the **scrsvPaint** method. Any value less than 100 ms (1/10th of a second) will be set at 100.

This method returns a String array whose elements correspond to those mentioned above. If this method is not defined, all this information will be empty and the frequency will be 1000 ms (1 second).

4. **scrsvInit** : This method, if it is declared, is called with launching of the screensaver before the first call to the method **scrsvPaint** and before **scrsvGetInfo**.

5. **scrsvFinish** : This method is called when the screensaver is interrupted.

6. **scrsvOpenConfig** : This method, if it is declared, is called when the user wants to configure the screensaver, or by right-clicking on the file ***JavaExe.scr*** (that is to say ***MyApp.scr***) then "Configure" in the contextual menu or from the control panel that manages screensavers then "Settings ...".

7. **scrsvPaint** (*Graphics2D* g, *int* wScr, *int* hScr) : This method is mandatory and will be called at regular intervals defined by **scrsvGetInfo** method.
   This method has three arguments :
   - ***g*** : corresponds to the graphics context used by some Java methods.
   - ***wScr*** : screen width in pixels.
   - ***hScr*** : screen height in pixels.

The dimensions of the screen always correspond to the dimensions of the desktop, even when viewing in preview mode in the control panel that manages screensavers. In preview mode, the dimensions match those of the desktop of the primary screen, in normal execution mode they correspond to the overall dimensions for all screens attached to the desktop.

8. **scrsvIsExitByKey** (*int* keycode, *boolean* isUp) : If this method is declared and returns **TRUE**, the screensaver will be interrupted when a key is pressed (or released).
   This method has 2 arguments :
   - ***keycode*** : code of the key pressed.
   - ***isUp*** : state of the key, pressed (**FALSE**) or released (**TRUE**).

9. **scrsvIsExitByMouse** (*int* x, *int* y, *int* nbClick, *int* button, *boolean* isUp) : If this method is declared and returns **TRUE**, the screensaver will stop when the mouse is moved or one of these buttons has been pressed (or released).
   This method has 5 arguments :
   - ***x*** and ***y*** : coordinates of the mouse pointer.
   - ***nbClick*** : number of clicks, if one of the mouse buttons is pressed, otherwise 0.
   - ***button*** : mouse button pressed: 1 = left, 2 = right, 3 = middle button if present..
   - ***isUp*** : state of the button, pressed (**FALSE**) or released (**TRUE**).

If neither of these methods interrupt is declared, **scrsvIsExitByKey** and **scrsvIsExitByMouse**, the screensaver will be interrupted at any mouse movement or any key is pressed.
See into Appendix, the ***JavaExe_I_ScreenSaverManagement*** interface and example 19.

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 "_ SystemEventManagement". For example, if my main class is called *MyApp*, then this method can be indifferently in *MyApp.class* or *MyApp_SystemEventManagement.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*>>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* & **ENDSESSION_LOGOFF**) != 0) is preferable with
      (*val2* == **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_APMQUERYSUSPEND
    PBT_APMQUERYSUSPENDFAILED
    PBT_APMSUSPEND
    PBT_APMRESUMECRITICAL
    PBT_APMRESUMESUSPEND
    PBT_APMBATTERYLOW
    PBT_APMPOWERSTATUSCHANGE
    PBT_APMOEMEVENT
    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 :
    - **SC_SCREENSAVE** : event on screensaver. Argument *val2* = **1** for starting, **0** for stopping.

    - **SC_MONITORPOWER** : event on monitor power. Argument *val2* = **-1** for power on, 2 for power off.

- **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_SystemEventManagement*** 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** (*boolean* isServiceUI); (or *void* **taskInit()** for previous versions)

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** ();

public static *void* **taskErrorNoService** ();

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. *LABEL* 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.
MFS_ADMIN = if the entry has the Admin icon (or according Windows).

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** (*boolean* isServiceUI) : This method is called at launching of the application. The ***isServiceUI*** argument is **TRUE** if this ***TaskbarManagement*** is part of an interactive service. However, for compatibility with previous versions of **JavaExe**, this same method is also accepted without argument

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.

15. **taskErrorNoService** : This method is called if the service has not been created.

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 regGetValueSTR (int hkey, String pathKey, String nameValue, boolean isExpandVal);
public static native byte[] regGetValueBIN (int hkey, String pathKey, String nameValue);
public static native int regGetValueDWORD (int hkey, String pathKey, String nameValue);
public static native long regGetValueQWORD (int hkey, String pathKey, String nameValue);
public static native String[] regGetValueMULTI (int hkey, String pathKey, String nameValue);

public static native boolean regSetValueSTR (int hkey, String pathKey, String nameValue, String val, boolean isTypeExpand);
public static native boolean regSetValueBIN (int hkey, String pathKey, String nameValue, byte[] val);
public static native boolean regSetValueDWORD (int hkey, String pathKey, String nameValue, int val
            ,boolean isTypeBigEndian);
public static native boolean regSetValueQWORD (int hkey, String pathKey, String nameValue, long val);
public static native boolean regSetValueMULTI (int hkey, String pathKey, String nameValue, String[] val);

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

public static native boolean regCreateKey (int hkey, String pathKey);
public static native boolean regDeleteKey (int hkey, String pathKey);
public static native boolean regDeleteValue (int hkey, String pathKey, String nameValue);

public static native String[] regEnumKeys (int hkey, String pathKey);
public static native String[] regEnumValues (int hkey, String pathKey);
```

In the previous version of ***JavaExe***, these methods were prefixed by "**nativeReg_**" and now they are by "**reg**", but the two notations are accepted.

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/en-us/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/en-us/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. **regGetValueSTR**: 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. **regGetValueBIN**: 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. **regGetValueDWORD**: 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. **regGetValueQWORD**: 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. **regGetValueMULTI**: 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. **regSetValueSTR**: 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. **regSetValueBIN**: 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. **regSetValueDWORD**: 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. **regSetValueQWORD**: 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. **regSetValueMULTI**: 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. **regGetTypeValue**: 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/en-us/library/ms724884.aspx for more information on the types in the Windows Registry.

Methods for the creation or deletion:

12. **regCreateKey**: 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. **regDeleteKey**: 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. **regDeleteValue**: 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. **regEnumKeys**: returns the list of subkeys contained directly in the path **pathKey**. The method will return `null` if the specified path is invalid.

16. **regEnumValues**: 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.

Dynamic Splash Screen

This feature allows the Java application to update dynamically the splash screen previously defined (see the section on static splash screen, page 20), and takes effect after displaying it.

Methods used as interface : *JavaExe_I_SplashScreenManagement*

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

It is not necessary all declare them, only those that the Java application needs.

1. public static *void* **sphInit** ();
2. public static *void* **sphFinish** ();
3. public static *boolean* **sphIsClose** ();   (or **isCloseSplash** for the previous versions of ***JavaExe***).
4. public static *int* **sphGetTickCount** ();
5. public static *String[]* **sphGetProgressBarInfo** ();
6. public static *int* **sphGetProgressBarValue** ();
7. public static *void* **sphPaint** (*Graphics2D* g, *int* wWnd, *int* hWnd);

1. **sphInit** : This method is called before any other methods of this feature, but after displaying the static splash screen.

2. **sphFinish** : This method will be called after closing the splash screen.

3. **sphIsClose** (or **isCloseSplash**) : This method, that must return **TRUE** to close the splash screen, is called at regular time interval (defined by the **sphGetTickCount** method). While it returns **FALSE**, the screen will remain visible.

4. **sphGetTickCount** : This method returns the refresh rate (in milliseconds), that is to say, the wait time between calls to the **sphPaint** method. Any value less than 100 ms (1/10th of a second) will be set at 100.
   If this method is not declared, the refresh rate is set to 1000 ms (1 second).

5. **sphGetProgressBarInfo** : If this method is declared, a progress bar will be displayed at the coordinates, relative to the splash screen, which will be returned by this method as a *String[]* :
   - **[0]** : X relative to the left edge of the splash screen.
   - **[1]** : Y relative to the top edge of the splash screen. If this value is negative, the positioning will be relative to the bottom edge minus the height of the bar defined in **[3]**.
   - **[2]** : W of the bar. If this value is negative, the width will be that of the splash screen minus X.
   - **[3]** : H of the bar. If this value is negative, it is set at 20 pixels by default.
   - **[4]** : maximum increment value of the progress bar. The current increment value symbolizes the progress and when this value reaches the maximum value, the splash screen will close.

If this method is not declared but **sphGetProgressBarValue** is, then the following values will be used by default: {0, -1, -1, 20, 10}. That is to say that the bar is positioned in the lower left corner, occupy the full width with a height of 20 pixels and the maximum increment is set at 10.

6. **sphGetProgressBarValue** : If this method is declared, a progress bar will be displayed at the coordinates returned by the **sphGetProgressBarInfo** method and will be called at regular time interval (defined by the

sphGetTickCount method) to return the current value of the increment of progress. When this value reaches the maximum value, the splash screen will close.

If this method is not declared (but **sphGetProgressBarInfo** is) the progress value is automatically incremented by 1 at each call.

If none of the two methods, **sphGetProgressBarInfo** and **sphGetProgressBarValue**, is declared then no progress bar will be displayed.

7. **sphPaint** (*Graphics2D* g, *int* wWnd, *int* hWnd) : This method will be called according to the refresh rate (defined by **sphGetTickCount**) to change the image of the splash screen, that its dimensions are **wWnd** by **hWnd** pixels. The argument **g** is the graphics context and will be pre-filled at each call with the image of the static splash screen.

   If the progress bar should be displayed, it will be after calling this method.

Admin/User Section Management

This feature allows the Java application to run in administrator mode, either another application, or itself but entirely or only a part of itself.

Methods used as interface : *JavaExe_I_SectionManagement*

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

It is not necessary all declare them, only those that the Java application needs.

Some of these methods are **native** type, that is to say, it is not necessary to define the methods body, but only their signature.

1. public static native *boolean* **sectionIsAdmin** ();
2. public static native *boolean* **sectionExecAsAdmin** (*String* pathname, *String[]* mainArgs);
3. public static native *void* **sectionRestartAsAdmin** (*Serializable* data, *String[]* mainArgs);
4. public static native *void* **sectionSetIconAdmin** (*Component* comp);
5. public static native *boolean* **sectionStartAdminThread** (*int* numID, *Serializable* data, *boolean* isWait);

6. public static *void* **sectionSetDataRestart** (*Serializable* data);

7. public static *void* **sectionMainAdmin** (*int* numID, *Serializable* data);
8. public static *void* **sectionClosedAdminThread** (*int* numID);

9. public static *boolean* **sectionIsDataForAdmin** (*int* numID);
10. public static *Serializable* **sectionDataForAdmin** (*int* numID);
11. public static *void* **sectionDataFromAdmin** (*int* numID, *Serializable* data);

12. public static *boolean* **sectionIsDataForUser** ();
13. public static *Serializable* **sectionDataForUser** ();
14. public static *void* **sectionDataFromUser** (*Serializable* data);

Methods to declare in native :

1. **sectionIsAdmin** : This method determines if the application is running in administrator mode or not.

2. **sectionExecAsAdmin** (*String* pathname, *String[]* mainArgs) : This method allows you to run another application in administrator mode. The **pathname** argument is the full path of the application including the name of the executable. The **mainArgs** argument contains the arguments that will be passed to the application. This method returns **`TRUE`** if the elevation in Admin mode been confirmed by the user and the application has been launched. This method does not wait for the execution of the application, it will return **`TRUE`** or **`FALSE`** immediately after the prompting for elevation.

3. **sectionRestartAsAdmin** (*Serializable* data, *String[]* mainArgs) : This method is used to restart the Java application in administrator mode with the same arguments of **main**() initially used to launch the application. This method has 2 arguments :

- ***data*** : object to send (or `null` if no object) to the application after it restarted in Admin mode before calling the **main**(). The application in Admin mode will receive this object from the method **sectionSetDataRestart**.
- ***mainArgs*** : list of additional arguments to be added to those that were used to initially start it in normal mode. These arguments (initial and additional) will be received by the **main**() of application in Admin mode.

This method applies only to a type of launch of Application or Service (for the interactive part only) and will have no effect on other types of launch.

4. **sectionSetIconAdmin** (*Component* comp) : This method allows to set, at a Swing or AWT component, the icon (or depending on the version of Windows) symbolizing the elevation request in administrator mode.

5. **sectionStartAdminThread** (*int* numID, *Serializable* data, *boolean* isWait) : This method allows you to run only a part of the Java application in administrator mode. In the remainder of this chapter, partial execution is called a ***AdminThread***. This method returns **`TRUE`** if the **numID** is not currently use and the elevation in Admin mode been confirmed by the user. It has 3 arguments :
   - ***numID*** : unique number (>= 0) to identify the ***AdminThread*** to perform.
   - ***data*** : data to send to the ***AdminThread*** at its launch.
   - ***isWait*** : If this argument is **`TRUE`**, calling this method will block until the end of the corresponding ***AdminThread***. It is important to note that this blocking also will impact the events of interaction with the user if this method is called from the main thread of the application. In this case it will be necessary to call this method from a secondary thread (see Example 14 for a specific case).

   During the call to this method, another will be called **sectionMainAdmin** (with the same arguments **numID** and **data**), which is the entry point of the partial execution in Admin mode, while the **main**() is the entry point for a full execution (admin or normal).

Restarting management in Admin mode :

6. **sectionSetDataRestart** (*Serializable* data) : This method is called before **main**(), when the Java application was restarted in Admin mode with **sectionRestartAsAdmin** method and receives the same **data** argument.

AdminThread Management :

7. **sectionMainAdmin** (*int* numID, *Serializable* data) : This method thus serves as an entry point to a partial execution in Admin mode, triggered by **sectionStartAdminThread** method and receives the same arguments, **numID** and **data**.

8. **sectionClosedAdminThread** (*int* numID) : This method is called, the side of the non-admin Java application, when the ***AdminThread*** matching to **numID** has completed execution.

Methods for communication between AdminThread and non-admin part :

9. **sectionIsDataForAdmin** (*int* numID) : This method returns **`TRUE`** if an object is available for the ***AdminThread*** matching to **numID**.

10. **sectionDataForAdmin** (*int* numID) : This method returns an object to the ***AdminThread*** matching to **numID**.

11. **sectionDataFromAdmin** (*int* numID, *Serializable* data) : This method will be called by the non-admin part with in argument an object from the ***AdminThread*** matching to **numID**.

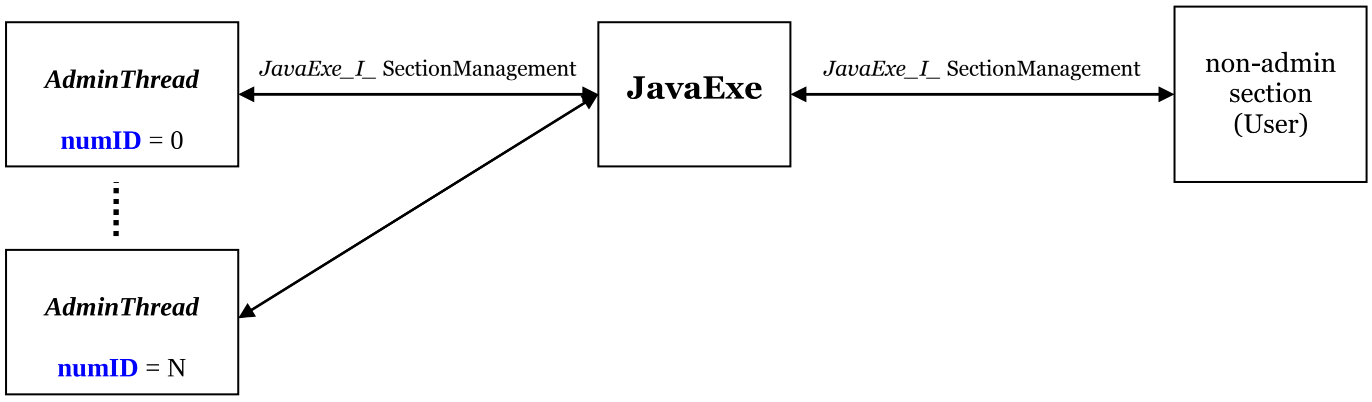
12. **sectionIsDataForUser** : This method returns **TRUE** if an object from the current ***AdminThread*** is available for non-admin part.

13. **sectionDataForUser** : This method returns an object for non-admin part from the current ***AdminThread***.

14. **sectionDataFromUser** (*Serializable* data) : This method will be called by the current ***AdminThread*** and receives in argument an object from the non-admin part.

Communication methods between ***AdminThread*** and non-admin section are based on the same principle (and therefore with the same constraints instances) for interactive services. That is to say that each ***AdminThread*** running in a different instance of the non-admin part.

Here is a diagram summarizing communication between ***AdminThread*** and non-admin section :



Service Control Management

This feature allows the Java application to manage any Windows services, that is to say, create, delete, start, stop, configure, ...

Methods used as interface : *JavaExe_I_ServiceControlManagement*

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

1. public static native *int* **scmCreateService** (*String* nameSvc, *String* nameLong, *String* descr, *String* pathnameExe, *boolean* isAuto ,*boolean* isDelayed);
2. public static native *int* **scmDeleteService** (*String* nameSvc);
3. public static native *int* **scmStartService** (*String* nameSvc);
4. public static native *int* **scmStopService** (*String* nameSvc);
5. public static native *int* **scmPauseService** (*String* nameSvc);
6. public static native *int* **scmContinueService** (*String* nameSvc);

7. public static native *int* **scmChangeConfig** (*String* nameSvc, *String* nameLong, *int* serviceType, *int* startType, *String* pathnameExe ,*String* dependencies, *String* login, *String* passwd);
8. public static native *int* **scmSetDescription** (*String* nameSvc, *String* descr);
9. public static native *int* **scmSetFailures** (*String* nameSvc, *String* actions, *String* delays, *String* exeFailure, *String* msgBoot ,*int* resetTime);
10. public static native *int* **scmSetDelayedAutoStart** (*String* nameSvc, *boolean* isDelayed);

11. public static native *String[][]* **scmEnumServices** (*int* serviceType, *int* serviceState, *boolean* isFullInfo);
12. public static native *String[][]* **scmEnumDependentServices** (*String* nameSvc, *int* serviceState, *boolean* isFullInfo);

13. public static native *String[]* **scmQueryConfig** (*String* nameSvc);
14. public static native *String[]* **scmQueryStatus** (*String* nameSvc);
15. public static native *String[]* **scmGetFailures** (*String* nameSvc);
16. public static native *String* **scmGetDescription** (*String* nameSvc);
17. public static native *String* **scmGetNameLong** (*String* nameSvc);
18. public static native *String* **scmGetNameSvc** (*String* nameLong);
19. public static native *boolean* **scmIsDelayedAutoStart** (*String* nameSvc);
20. public static native *String* **scmGetErrorMessage** (*int* numErr);

Generally, the **nameSvc** arguments correspond to the service short name (usually the name of the .exe file), Each service is identified by this name. While **nameLong** arguments correspond to a long name, that is to say a short description.

The **nameSvc** arguments can be `null` if and only if the Java application that calls these methods is recognized as a service. In this case **nameSvc** matches the name .exe file, but without the extension.

Methods of creation / deletion :

1. **scmCreateService** (*String* nameSvc, *String* nameLong, *String* descr, *String* pathnameExe, *boolean* isAuto, *boolean* isDelayed) :
   This method allows you to create a service that will be identified by **nameSvc**, whose path to the executable file is indicated by **pathnameExe** (including the name of the .exe itself). Once the service has been created, its **nameSvc** cannot be changed. This method returns 0 if successful, otherwise an error number.
   The other arguments correspond to :
   - **descr** : long description of the service.
   - **isAuto** : `TRUE` if the service should be started automatically with the system.
   - **isDelayed** : `TRUE` if the service will be started in delayed mode (it means that it should start automatically), that is to say, it will be started after all automatic services non-delayed are started.

2. **scmDeleteService** (*String* nameSvc) :
   This method removes a service identified by **nameSvc**. If this service is running, it will not be stopped but marked "For Deleting" (that is to say that its startup type is set to *SERVICE_DISABLED*) and will actually be deleted when the system restarts, or when the service will be stopped.
   This method returns 0 if successful, otherwise an error number.

Methods to change the status :

3. **scmStartService** (*String* nameSvc) : This method is used to start a stopped service identified by **nameSvc** and returns 0 if successful, otherwise an error number.

4. **scmStopService** (*String* nameSvc) : This method stops a service, running or paused, identified by **nameSvc** and returns 0 if successful, otherwise an error number.

5. **scmPauseService** (*String* nameSvc) : This method allows you to pause a running service identified by **nameSvc** and returns 0 if successful, otherwise an error number.

6. **scmContinueService** (*String* nameSvc) : This method allows to continue the execution of a paused service, identified by **nameSvc** and returns 0 if successful, otherwise an error number.

Methods to configure :

7. **scmChangeConfig** (*String* nameSvc, *String* nameLong, *int* serviceType, *int* startType, *String* pathnameExe, *String* dependencies, *String* login, *String* passwd) :
   This method allows you to change certain attributes of the basic configuration of a service, identified by **nameSvc**, such as :
   - **nameLong** : new long name (short description) of the service, or `null` if no change.
   - **serviceType** : new type of service (see the possible values in the Appendix), or *SERVICE_NO_CHANGE* if no change.
   - **startType** : new startup type (see possible values in the Appendix), or *SERVICE_NO_CHANGE* if no change.
   - **pathnameExe** : new path to the executable file, or `null` if no change.
   - **dependencies** : new dependencies of the service, or `null` if no change.
   - **login** : new login used by the service, or `null` if no change.
   - **passwd** : new password (associated with the login), or `null` if no change.

   This method returns 0 if successful, otherwise an error number.

8. **scmSetDescription** (*String* nameSvc, *String* descr) :
   This method allows you to change the long description of a service identified by **nameSvc**. If **descr** argument is `null`, no changes will be made. This method returns 0 if successful, otherwise an error number.

9. **scmSetFailures** (*String* nameSvc, *String* actions, *String* delays, *String* exeFailure, *String* msgBoot, *int* resetTime) :

   This method allows you to change the failure actions of the service identified by **nameSvc** :

   - **actions** : list of actions (see page 21 for more details), or `null` if no change.
   - **delays** : List of delays (see page 21 for more details), or `null` if no change.
   - **exeFailure** : path to an executable file (including its arguments) in case of failure when the action list contains *RUN*, or `null` if no change.
   - **msgBoot** : message will be displayed on the computer when the action list contains *REBOOT*, or `null` if no change.
   - **resetTime** : delays to reset the actions counter (see page 21 for more details).

   This method returns 0 if successful, otherwise an error number.

10. **scmSetDelayedAutoStart** (*String* nameSvc, *boolean* isDelayed) :

    This method allows you to change the attribute of the automatic delayed of service identified by **nameSvc**. If **isDelayed** argument is `TRUE`, the service will launch with delay (this implies that it is configured to start automatically).

    This method returns 0 if successful, otherwise an error number.

Enumeration of services :

11. **scmEnumServices** (*int* serviceType, *int* serviceState, *boolean* isFullInfo) :

    This method provides a list of services that meet certain criteria, such as :

    - **serviceType** : filter by type of service : *SERVICE_WIN32* for application services, *SERVICE_DRIVER* for system kernel services, *SERVICE_TYPE_ALL* for all services (see Appendix for all values).
    - **serviceState** : filter according to the status of the service: *SERVICE_ACTIVE* for active service, *SERVICE_INACTIVE* for inactive, *SERVICE_STATE_ALL* for both.

    The result will be an array in which each element corresponds to a single service corresponding to the criteria and contains information (as *String[]*) on this service, such as :

    - **[0]** : short name of the service (**nameSvc**).
    - **[1]** : long name of service (short description).
    - **[2]** : service status : *SERVICE_STOPPED* (1), *SERVICE_RUNNING* (4), *SERVICE_PAUSED* (7), … (see the possible values in Appendix).

    Then if **isFullInfo** argument is `TRUE`, more information will be available :

    - **[3]** : service type : *SERVICE_KERNEL_DRIVER* (1), *SERVICE_WIN32_OWN_PROCESS* (16), …
    - **[4]** : accepted control codes : *SERVICE_ACCEPT_STOP* (1) service accepts to be stopped, *SERVICE_ACCEPT_PAUSE_CONTINUE* (2) service accepts to be paused, …
    - **[5]** : startup type : *SERVICE_AUTO_START* (2) automatic startup, *SERVICE_DEMAND_START* (3) manual startup, …
    - **[6]** : value 1 if the automatic startup is delayed, otherwise 0.
    - **[7]** : long description.

12. **scmEnumDependentServices** (*String* nameSvc, *int* serviceState, *boolean* isFullInfo) :

    This method provides a list of services that depend on the service identified by **nameSvc**. These services can be enabled or disabled according to the argument **serviceState**. The result of this method is the same as before, with the method **scmEnumServices**.

Information Request :

13. **scmQueryConfig** (*String* nameSvc) : This method provides some configuration information for a service identified by **nameSvc**. The result is a *String[]* whose values are :
    - **[0]** : long name of service (short description).
    - **[1]** : pathname of the executable file.
    - **[2]** : dependencies of the service (services list separated by "/")..
    - **[3]** : login used by the service.
    - **[4]** : service type : *SERVICE_WIN32_OWN_PROCESS* (16), …
    - **[5]** : startup type : *SERVICE_AUTO_START* (2), …

14. **scmQueryStatus** (*String* nameSvc) : This method provides some status information of a service identified by **nameSvc**. The result is a *String[]* whose values are :
    - **[0]** : current status : *SERVICE_STOPPED* (1), *SERVICE_RUNNING* (4), …
    - **[1]** : service type : *SERVICE_WIN32_OWN_PROCESS* (16), …
    - **[2]** : accepted control codes : *SERVICE_ACCEPT_STOP* (1) , …
    - **[3]** : PID number if the service is running.

15. **scmGetFailures** (*String* nameSvc) : This method provides some information on actions failure of a service identified by **nameSvc**. The result is a *String[]* whose values are :
    - **[0]** : actions list separated by "/" (for more details, see page XX).
    - **[1]** : delays list separated by "/" (for more details, see page XX).
    - **[2]** : pathname to executable file (including its arguments) in case of failure when the action list contains *RUN*.
    - **[3]** : message will be displayed on the computer when the action list contains *REBOOT*.
    - **[4]** : period reset of the failure actions counter.

16. **scmGetDescription** (*String* nameSvc) : This method provides a long description of a service identified by **nameSvc**.

17. **scmGetNameLong** (*String* nameSvc) : This method provides the long name of a service identified by **nameSvc**.

18. **scmGetNameSvc** (*String* nameLong) : This method provides the short name (**nameSvc**) of a service identified by its long name **nameLong**.

19. **scmIsDelayedAutoStart** (*String* nameSvc) : This method returns `TRUE` if the service identified by **nameSvc** is configured to be launched in automatic delayed.

20. **scmGetErrorMessage** (*int* numErr) : This method returns the error message corresponding to the error number **numErr**. This message depends on the language installed by default on Windows.

System Management

This feature allows the Java application to call some Windows systems functions such as standby, shutdown or restart the system, or the user log off, …

Methods used as interface : *JavaExe_I_SystemManagement*

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

It is not necessary all declare them, only those that the Java application needs.

These methods are **native** type, that is to say that it is not necessary to define the method bodies, but only their signature.

1. public static native int **systemShutdown** (*String* msg, *int* timeOut, *boolean* isReboot, *boolean* isForce, *boolean* isPlanned);
2. public static native int **systemAbortShutdown** ();
3. public static native int **systemOpenDialogShutdown** (*boolean* isReboot, *boolean* isForce);
4. public static native int **systemStandby** (*boolean* isHibernate, *boolean* isDisableWake);
5. public static native int **systemLock** ();
6. public static native int **systemUserLogoff** (*boolean* isForce);
7. public static native int **systemOpenDialogLogoff** (*boolean* isForce);
8. public static native int **systemSetRequiredState** (*boolean* isNoScreenSaver, *boolean* isNoDisplayOff, *boolean* isNoStandby);
9. public static native int **systemBlockShutdown** (*String* reason);
10. public static native int **systemUnblockShutdown** ();
11. public static native boolean **systemIsLocked** ();
12. public static native boolean **systemIsShutdownAllowed** ();
13. public static native boolean **systemIsHibernateAllowed** ();
14. public static native boolean **systemIsStandbyAllowed** ();
15. public static native String **systemGetErrorMessage** (*int* numErr);

Methods of shutdown and restarting :

1. **systemShutdown** (*String* msg, *int* timeOut, *boolean* isReboot, *boolean* isForce, *boolean* isPlanned) :
   This method is used to trigger the shutdown, possibly followed by a reboot if the **isReboot** argument is **TRUE**. The other arguments are :
   - **msg** : message will be displayed for a while (defined by **timeOut** argument, if greater than 0), or **null** if no message.
   - **timeOut** : delay (in seconds) before triggering of the shutdown, or 0 for an immediate shutdown.
   - **isForce** : **TRUE** to force applications to quit without waiting for confirmation.
   - **isPlanned** : **TRUE** to indicate that this is a planned shutdown, that is to say not unexpected. Some versions of Windows will display a warning when the system is restarted after an unexpected shutdown.

   This method returns 0 if successful, otherwise an error number.

2. **systemAbortShutdown** : This method allows you to cancel a request for non immediate shutdown and returns 0 if successful, otherwise an error number.

3. **systemOpenDialogShutdown** (*boolean* isReboot, *boolean* isForce) : This method opens a Windows dialog box asking the user whether to shutdown or restart the system (depending on the argument **isReboot**). The other argument (**isForce**) is the same as **systemShutdown**. This method returns 0 if successful, otherwise an error number.

Method of standby :

4. **systemStandby** (*boolean* isHibernate, *boolean* isDisableWake) :
   This method allows the system to standby or hibernation if the argument **isHibernate** is **TRUE**. If **isDisableWake** argument is **TRUE**, the system standby cannot be awakened by automatic triggering events, such as WindowsUpdate example. It returns 0 if successful, otherwise an error number.

Methods for the logged on user :

5. **systemLock** : This method allows the user currently logged on to lock the computer and returns 0 if successful, otherwise an error number.

6. **systemUserLogoff** (*boolean* isForce) : This method allows the user currently logged on, to disconnect and returns 0 if successful, otherwise an error number. If **isForce** argument is **TRUE**, the applications will be forced to leave without waiting for confirmation.

7. **systemOpenDialogLogoff** (*boolean* isForce) : This method opens a Windows dialog box asking if the user wants to log off now or later, and returns 0 if successful, otherwise an error number. The argument **isForce** is the same as before.

Blocking methods :

8. **systemSetRequiredState** (*boolean* isNoScreenSaver, *boolean* isNoDisplayOff, *boolean* isNoStandby) :
   This method prevents the running of the screensaver (if **isNoScreenSaver** is **TRUE**) or monitor off (if **isNoDisplayOff** is **TRUE**), or automatic standby (if **isNoStandby** is **TRUE**). This method returns 0 if successful, otherwise an error number.

9. **systemBlockShutdown** (*String* reason) : This method will block a shutdown. The reason for this block is provided by the argument **reason** and will be displayed on some versions of Windows. This method returns 0 if successful, otherwise an error number.

10. **systemUnblockShutdown** : This method cancels the previous blocking and re-authorizes the shutdown. It returns 0 if successful, otherwise an error number.

Information request :

11. **systemIsLocked** : This method returns **TRUE** if the computer is locked. It will always return **FALSE** if it is called from a service (except for its interactive part).

12. **systemIsShutdownAllowed** : This method returns **TRUE** if Windows allows a system shutdown.
13. **systemIsHibernateAllowed** : This method returns **TRUE** if Windows allows hibernation.
14. **systemIsStandbyAllowed** : This method returns **TRUE** if Windows allows standby.

15. **systemGetErrorMessage** (*int* numErr) : This method returns the error message corresponding to the error number **numErr**. This message depends on the language installed by default on Windows.

Appendices

Java Interfaces

These interfaces aren't used by **JavaExe** but are provided as a guide only to have access to the constants needed in different features. Class methods defined therein are just a reminder since the methods defined in an interface apply to instances.

ApplicationManagement

```
interface  JavaExe_I_ApplicationManagement
{
        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 void cplOpen ();

        public static String[] cplGetInfo ();
}
```

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 regGetValueSTR (int hkey, String pathKey, String nameValue
                    ,boolean isExpandVal);
        public static native byte[] regGetValueBIN (int hkey, String pathKey, String nameValue);
        public static native int regGetValueDWORD (int hkey, String pathKey, String nameValue);
        public static native long regGetValueQWORD (int hkey, String pathKey, String nameValue);
        public static native String[] regGetValueMULTI (int hkey, String pathKey, String nameValue);

        public static native boolean regSetValueSTR (int hkey, String pathKey, String nameValue, String val
                    ,boolean isTypeExpand);
        public static native boolean regSetValueBIN (int hkey, String pathKey, String nameValue, byte[] val);
        public static native boolean regSetValueDWORD (int hkey, String pathKey, String nameValue, int val
                    ,boolean isTypeBigEndian);
        public static native boolean regSetValueQWORD (int hkey, String pathKey, String nameValue
                    ,long val);
        public static native boolean regSetValueMULTI (int hkey, String pathKey, String nameValue
                    ,String[] val);

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

        public static native boolean regCreateKey (int hkey, String pathKey);
        public static native boolean regDeleteKey (int hkey, String pathKey);
        public static native boolean regDeleteValue (int hkey, String pathKey, String nameValue);

        public static native String[] regEnumKeys (int hkey, String pathKey);
        public static native String[] regEnumValues (int hkey, String pathKey);
}
```

ScreenSaverManagement

```
interface  JavaExe_I_ScreenSaverManagement
{
        public static boolean scrsvIsCreate ();
        public static boolean scrsvIsDelete ();

        public static String[] scrsvGetInfo ();
        public static void scrsvInit ();
        public static void scrsvFinish ();

        public static void scrsvOpenConfig ();
        public static void scrsvPaint (Graphics2D g, int wScr, int hScr);

        public static boolean scrsvIsExitByKey (int keycode, boolean isUp);
        public static boolean scrsvIsExitByMouse (int x, int y, int nbClick, int button, boolean isUp);
}
```

SectionManagement

```
interface  JavaExe_I_SectionManagement
{
        public static native boolean sectionIsAdmin ();
        public static native boolean sectionExecAsAdmin (String pathname, String[] mainArgs);
        public static native void sectionRestartAsAdmin (Serializable data, String[] mainArgs);
        public static native void sectionSetIconAdmin (Component comp);
        public static native boolean sectionStartAdminThread (int numID, Serializable data, boolean isWait);

        public static void sectionSetDataRestart (Serializable data);

        public static void sectionMainAdmin (int numID, Serializable data);
        public static void sectionClosedAdminThread (int numID);

        public static boolean sectionIsDataForAdmin (int numID);
        public static Serializable sectionDataForAdmin (int numID);
        public static void sectionDataFromAdmin (int numID, Serializable data);

        public static boolean sectionIsDataForUser ();
        public static Serializable sectionDataForUser ();
        public static void sectionDataFromUser (Serializable data);
}
```

ServiceControlManagement

```
interface  JavaExe_I_ServiceControlManagement
{
        //--- Service Type
        static final int SERVICE_KERNEL_DRIVER              = 0x00000001;
        static final int SERVICE_FILE_SYSTEM_DRIVER         = 0x00000002;
        static final int SERVICE_ADAPTER                    = 0x00000004;
        static final int SERVICE_RECOGNIZER_DRIVER          = 0x00000008;
        static final int SERVICE_WIN32_OWN_PROCESS          = 0x00000010;
        static final int SERVICE_WIN32_SHARE_PROCESS        = 0x00000020;
        static final int SERVICE_INTERACTIVE_PROCESS        = 0x00000100;
        static final int SERVICE_WIN32                      = (SERVICE_WIN32_OWN_PROCESS
                | SERVICE_WIN32_SHARE_PROCESS);
        static final int SERVICE_DRIVER                     = (SERVICE_KERNEL_DRIVER
                | SERVICE_FILE_SYSTEM_DRIVER | SERVICE_RECOGNIZER_DRIVER);
        static final int SERVICE_TYPE_ALL                   = (SERVICE_WIN32 | SERVICE_ADAPTER
                | SERVICE_DRIVER | SERVICE_INTERACTIVE_PROCESS);

        //--- Start Type
        static final int SERVICE_BOOT_START            = 0x00000000;
        static final int SERVICE_SYSTEM_START        = 0x00000001;
        static final int SERVICE_AUTO_START          = 0x00000002;
        static final int SERVICE_DEMAND_START        = 0x00000003;
        static final int SERVICE_DISABLED            = 0x00000004;

        //--- Current Status
        static final int SERVICE_STOPPED             = 0x00000001;
        static final int SERVICE_START_PENDING       = 0x00000002;
        static final int SERVICE_STOP_PENDING        = 0x00000003;
        static final int SERVICE_RUNNING             = 0x00000004;
        static final int SERVICE_CONTINUE_PENDING= 0x00000005;
        static final int SERVICE_PAUSE_PENDING       = 0x00000006;
        static final int SERVICE_PAUSED              = 0x00000007;

        //--- Service State
        static final int SERVICE_ACTIVE        = 0x00000001;
        static final int SERVICE_INACTIVE      = 0x00000002;
        static final int SERVICE_STATE_ALL     = (SERVICE_ACTIVE | SERVICE_INACTIVE);

        //--- Controls Code
        static final int SERVICE_ACCEPT_STOP                              = 0x00000001;
        static final int SERVICE_ACCEPT_PAUSE_CONTINUE                    = 0x00000002;
        static final int SERVICE_ACCEPT_SHUTDOWN                          = 0x00000004;
        static final int SERVICE_ACCEPT_PARAMCHANGE                       = 0x00000008;
        static final int SERVICE_ACCEPT_NETBINDCHANGE                     = 0x00000010;
        static final int SERVICE_ACCEPT_HARDWAREPROFILECHANGE             = 0x00000020;
        static final int SERVICE_ACCEPT_POWEREVENT                        = 0x00000040;
        static final int SERVICE_ACCEPT_SESSIONCHANGE                     = 0x00000080;
        static final int SERVICE_ACCEPT_PRESHUTDOWN                       = 0x00000100;

        //--- Error Code
        static final int ERROR_SUCCESS                             = 0;
        static final int ERROR_PATH_NOT_FOUND                      = 3;
        static final int ERROR_ACCESS_DENIED                       = 5;
        static final int ERROR_INVALID_NAME                        = 123;
        static final int ERROR_DEPENDENT_SERVICES_RUNNING          = 1051;
        static final int ERROR_INVALID_SERVICE_CONTROL             = 1052;
        static final int ERROR_SERVICE_REQUEST_TIMEOUT             = 1053;
        static final int ERROR_SERVICE_NO_THREAD                   = 1054;
        static final int ERROR_SERVICE_DATABASE_LOCKED             = 1055;
        static final int ERROR_SERVICE_ALREADY_RUNNING             = 1056;
        static final int ERROR_INVALID_SERVICE_ACCOUNT             = 1057;
        static final int ERROR_SERVICE_DISABLED                    = 1058;
        static final int ERROR_CIRCULAR_DEPENDENCY                 = 1059;
        static final int ERROR_SERVICE_DOES_NOT_EXIST              = 1060;
        static final int ERROR_SERVICE_CANNOT_ACCEPT_CTRL          = 1061;
        static final int ERROR_SERVICE_NOT_ACTIVE                  = 1062;
        static final int ERROR_SERVICE_DEPENDENCY_FAIL             = 1068;
        static final int ERROR_SERVICE_LOGON_FAILED                = 1069;
        static final int ERROR_SERVICE_MARKED_FOR_DELETE           = 1072;
        static final int ERROR_SERVICE_EXISTS                      = 1073;
        static final int ERROR_SERVICE_DEPENDENCY_DELETED          = 1075;
        static final int ERROR_DUPLICATE_SERVICE_NAME              = 1078;
        static final int ERROR_SHUTDOWN_IN_PROGRESS                = 1115;

        //--- Change Config
        static final int SERVICE_NO_CHANGE = -1;

        //--- Result QueryConfig
        static final int NDX_CONFIG_NAMELONG         = 0;
        static final int NDX_CONFIG_PATHNAME         = 1;
        static final int NDX_CONFIG_DEPENDS          = 2;
        static final int NDX_CONFIG_LOGIN            = 3;
        static final int NDX_CONFIG_TYPE             = 4;
        static final int NDX_CONFIG_START            = 5;
```

```
        //--- Result QueryStatus
        static final int NDX_STATUS_CURRENT          = 0;
        static final int NDX_STATUS_TYPE             = 1;
        static final int NDX_STATUS_CNTRL            = 2;
        static final int NDX_STATUS_PRCSSID          = 3;

        //--- Result Enum Services
        static final int NDX_ENUM_NAMESVC            = 0;
        static final int NDX_ENUM_NAMELONG           = 1;
        static final int NDX_ENUM_STATUS             = 2;
        static final int NDX_ENUM_TYPE               = 3;
        static final int NDX_ENUM_CNTRL              = 4;
        static final int NDX_ENUM_START              = 5;
        static final int NDX_ENUM_DELAYED            = 6;
        static final int NDX_ENUM_DESCR              = 7;


        /*******************************************/
        public static native int scmCreateService (String nameSvc, String nameLong, String descr, String pathnameExe
                ,boolean isAuto, boolean isDelayed);
        public static native int scmDeleteService (String nameSvc);
        public static native int scmStartService (String nameSvc);
        public static native int scmStopService (String nameSvc);
        public static native int scmPauseService (String nameSvc);
        public static native int scmContinueService (String nameSvc);

        public static native int scmChangeConfig (String nameSvc, String nameLong, int serviceType, int startType
                ,String pathnameExe, String dependencies, String login, String passwd);
        public static native int scmSetDescription (String nameSvc, String descr);
        public static native int scmSetFailures (String nameSvc, String actions, String delays, String exeFailure
                ,String msgBoot, int resetTime);
        public static native int scmSetDelayedAutoStart (String nameSvc, boolean isDelayed);

        public static native String[][] scmEnumServices (int serviceType, int serviceState, boolean isFullInfo);
        public static native String[][] scmEnumDependentServices (String nameSvc, int serviceState
                ,boolean isFullInfo);

        public static native String[] scmQueryConfig (String nameSvc);
        public static native String[] scmQueryStatus (String nameSvc);
        public static native String[] scmGetFailures (String nameSvc);
        public static native String scmGetDescription (String nameSvc);
        public static native String scmGetNameLong (String nameSvc);
        public static native String scmGetNameSvc (String nameLong);
        public static native boolean scmIsDelayedAutoStart (String nameSvc);

        public static native String scmGetErrorMessage (int numErr);
}
```

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 ();
}
```

SplashScreenManagement

```
interface  JavaExe_I_SplashScreenManagement
{
        public static void sphInit ();
        public static void sphFinish ();

        public static boolean sphIsClose ();
        public static int sphGetTickCount ();

        public static String[] sphGetProgressBarInfo ();
        public static int sphGetProgressBarValue ();

        public static void sphPaint (Graphics2D g, int wWnd, int hWnd);
}
```

SystemEventManagement

```
interface  JavaExe_I_SystemEventManagement
{
        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_APMSUSPEND                      = 0x0004;
        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_DEVICETYPESPECIFIC         = 0x8005;
        static final int DBT_CUSTOMEVENT                = 0x8006;
        static final int DBT_USERDEFINED                = 0xFFFF;

        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 SC_MONITORPOWER        = 0xF170;

        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);
}
```

SystemManagement

```
interface  JavaExe_I_SystemManagement
{
        //--- Error Code
        static final int ERROR_SUCCESS                          = 0;
        static final int ERROR_ACCESS_DENIED                    = 5;
        static final int ERROR_NOT_SUPPORTED                    = 50;
        static final int ERROR_FAIL_SHUTDOWN                    = 351;
        static final int ERROR_SYSTEM_SHUTDOWN                  = 641;
        static final int ERROR_SHUTDOWN_IN_PROGRESS             = 1115;
        static final int ERROR_NO_SHUTDOWN_IN_PROGRESS        = 1116;
        static final int ERROR_SHUTDOWN_IS_SCHEDULED          = 1190;
        static final int ERROR_SHUTDOWN_USERS_LOGGED_ON       = 1191;
        static final int ERROR_SERVER_SHUTDOWN_IN_PROGRESS  = 1255;


        /*******************************************************/
        public static native int systemShutdown (String msg, int timeOut, boolean isReboot, boolean isForce
                ,boolean isPlanned);
        public static native int systemAbortShutdown ();
        public static native int systemOpenDialogShutdown (boolean isReboot, boolean isForce);

        public static native int systemStandby (boolean isHibernate, boolean isDisableWake);

        public static native int systemLock ();
        public static native int systemUserLogoff (boolean isForce);
        public static native int systemOpenDialogLogoff (boolean isForce);

        public static native int systemSetRequiredState (boolean isNoScreenSaver, boolean isNoDisplayOff
                ,boolean isNoStandby);
        public static native int systemBlockShutdown (String reason);
        public static native int systemUnblockShutdown ();

        public static native boolean systemIsLocked ();
        public static native boolean systemIsShutdownAllowed ();
        public static native boolean systemIsHibernateAllowed ();
        public static native boolean systemIsStandbyAllowed ();
        public static native String systemGetErrorMessage (int numErr);
}
```

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   = 0x00000003;
        static final int MFS_CHECKED    = 0x00000008;
        static final int MFS_HILITE         = 0x00000080;
        static final int MFS_ADMIN          = 0x10000000;


        /*******************************************/
        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 (boolean isServiceUI);

            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 ();

            public static void taskErrorNoService ();
        }
```

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 (in admin mode) for install / uninstall the control panel, or in double clicking on the .CPL to open it directly.

The control panel will be installed in two categories: "**Appearance and Personalization**" and "**Security**".

3 - Service

This example define a service without Desktop interaction. At the time of its installation (in admin mode), 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.

12 - Run as Admin 1

The executable of this example contains a manifest forcing Windows to run automatically in administrator mode. The manifest was added to .exe with **UpdateRsrcJavaExe**.

In this example, the button will also launch CMD.EXE in admin mode.

13 - Run as Admin 2

This example runs in normal mode (that is to say in the context of the current user) and contains two buttons each with an icon (or depending on the version of Windows) symbolizing the elevation request in administrator mode. The first button "**Restart App**" restarts the same example in admin mode if the elevation request has been confirmed. The second button "**Execute CMD.EXE**" launchs a DOS window (CMD.EXE) in admin mode if the elevation is confirmed.

14 - Thread as Admin 1

This example shows the ***AdminThread*** functionality to execute only a part of the Java application in admin mode, while the rest is still in standard mode.

In this example, there is a "**Run Thread**" button to start a new thread in admin mode (after confirmation of the elevation request), and a "**File**" menu which it is possible to run an action in Admin mode or standard mode.

15 - Thread as Admin 2

This example shows the ***AdminThread*** functionality to execute only a part of the Java application in admin mode, and the communication with these ***AdminThread***.

In this example, there is a "**New Thread**" button to start a new thread in admin mode (after confirmation of the elevation request). When at least one ***AdminThread*** is active, it is possible to send a text and vice versa.

16 - Dynamic SplashScreen 1

This example shows the dynamic splash screen functionality, in which the static splash screen is updated with an initialization text that increments for 5 seconds.

17 - Dynamic SplashScreen 2

This example shows the dynamic splash screen functionality, in which the static splash screen is updated with a progress bar for 5 seconds.

18 - Dynamic SplashScreen 3

This example combines the previous two: the static screen is updated with a text showing the initialization step and a progress bar.

19 - ScreenSaver

This example illustrates the screen saver feature. Right-click on the file "**ScreenSaver.scr**" can either test it, or open its configuration screen, or install it.

It is also possible to install it directly from the .exe, but in administrator mode, or uninstall it if already installed.

To stop this screensaver, simply press ESC key or double-click the right mouse button.

20 - SystemManagement

This example provides access to all the features controlling the stop (or restart) the PC, its standby or hibernate, disconnect the current user, locking the station, or prevents trigger a screensaver, or the monitor off, or automatic standby (that is to say, not requested by the user).

21 – ServiceControlManagement

This example put into practice the features of the Service Control Management, that is to say the ability to manage Windows services. It shows the various application services and / or those of the system kernel, and details of their current status and configuration.

By clicking on the button "**Change ...**" it is possible to change the startup type, or stop the service, pause it ... But these changes will be allowed if this example has been launched in administrator mode.

22 - ServiceControlManagement & Admin

Same example as above, but this time by clicking on the button "**Change ...**" prompting for elevation to administrator mode to be authorized to make these changes.

23 - Service & TrayIcon & System Event & SCM & Admin

This example is the same as Example 8, except that it is now possible, from menu of the icon, stop the service or restart with elevation request in admin mode, if the part IU of the service had not been launched in admin mode (in which case it will be also possible to restart the UI part in admin mode).