

JavaExe

version 3.0.1



par DevWizard
(DevWizard@free.fr)

(30 Octobre 2006)

à Jawaher...

Table des matières

Présentation.....	4
Historique.....	5
Utilisation générale.....	6
Création du .EXE.....	6
Les propriétés.....	7
Utilisation de UpdateRsrcJavaExe (ex MergeICO).....	9
Changer l'icône du .EXE.....	10
Ecran de démarrage (le Screen Splash).....	10
Lancement en tant qu'application.....	11
Lancement en tant que service.....	12
Lancement en tant que Panneau de contrôle.....	15
Fonctionnalités additionnelles	
Gestion des Evénements Système.....	18
Gestion de la barre des tâches.....	22
Annexes	
Interfaces Java.....	27
ApplicationManagement.....	27
ControlPanelManagement.....	27
ServiceManagement.....	27
SystemEventManager.....	28
TaskbarManagement.....	29
Exemples.....	31
1 – Application.....	31
2 - Control Panel.....	31
3 – Service.....	31
4 – TrayIcon.....	31
5 - Service & TrayIcon.....	32
6 - System Event.....	32
7 – OneInstance.....	32
8 - Service & TrayIcon & System Event.....	32

Présentation

JavaExe permet de lancer votre application Java à partir d'un exécutable comme s'il s'agissait d'une application Windows, ou d'un service système ou encore en tant que Panneau de configuration (Control Panel).

« L'imagination est plus importante que la connaissance »
Albert Einstein

Historique

- **3.0.1** (30 Octobre 2006) :
 - Correction d'un bug mineur dans **UpdateRsrcJavaExe** : les fichiers associés aux coches étaient toujours pris en compte même si la coche correspondante n'était pas sélectionnée.
 - Correction d'un bug majeur concernant l'exemple « 7 – OneInstance » : le résultat de la méthode **isOneInstance** n'était pas toujours pris en compte dans certaine version de Windows XP, et l'exemple « 8 - Service & TrayIcon & System Event » : la partie interactive ne se lançait pas dans tous les cas.
 - Le numéro de version minimum requis de Java est indiqué dans le message d'alerte si aucun JRE n'est trouvé.
- **3.0** (11 Septembre 2006) :
 - Gestion d'une application Java en tant que Panneau de contrôle (avec le fichier **JavaExe.cpl**)
 - Gestion de la barre des tâches et de son icône.
 - Gestion des événements systèmes.
 - Affichage d'un écran de démarrage avant le lancement de l'application Java.
 - Possibilité de contrôler le nombre d'instance en cours d'exécution de la même application.
 - Renommage de l'outil **MergeICO** en **UpdateRsrcJavaExe**.
 - Properties :
 - Ajout de *URL_InstallJRE*, *PathJRE*, *PathBrowser*, *Display_BoxInstall*
 - RunAsService* : est renommée en *RunType*
 - RunType* : ajout d'un type (2) pour le mode ControlPanel.
 - ClassDirectory* : est mis par défaut à « resource »
 - Lecture du « manifest » du .jar principal pour trouver automatiquement la classe principale.
- **2.0** (16 Novembre 2003) :
 - Lancement de l'application Java directement avec la JVM si possible. Sinon lancement via la commande **java.exe**
 - Possibilité de lancer l'application comme un service Windows.
 - Création d'un 2^{ème} fichier exécutable nommé **JavaExe_console.exe** pour lancer l'application avec une console DOS.
 - Ajout de quelques propriétés : *ClassDirectory*, *PersonalOptions*, *ResourceDirectory*, *RunAsService*
 - La propriété *JREversion* signifie maintenant la version minimum au lieu de la version stricte.
- **1.3** (21 Avril 2003) :
 - Correction d'un bug potentiel dans **JavaExe.exe** (un pb lié aux "\" dans la variable de propriétés *PersonalClasspath*)
- **1.2** (4 Novembre 2002) :
 - Correction d'un bug dans **MergeICO.exe** (le déplacement d'une icône sur **MergeICO.exe** n'était pas pris en compte)
 - Lancement de l'application Java avec le paramètre *java.library.path* fixé à ".;\resource\", vous permettant ainsi de mettre vos éventuelles DLL (pour les méthodes natives par exemple) dans le même répertoire que votre application ou dans le répertoire "resource".
- **1.1** (5 Octobre 2002) :
 - Ajout d'une propriété, **Main Class**, dans le fichier **JavaExe.properties**. Cette propriété est nécessaire lorsque la classe principale se trouve dans un package.
- **1.0** (28 Août 2002) : version initiale.

Utilisation générale

Création du .EXE

Pour obtenir un fichier exécutable de votre application Java, il suffit tout simplement de copier le fichier **JavaExe.exe** dans votre répertoire contenant l'application Java, puis lui donner le même nom que votre classe principale. **JavaExe.exe** est fourni avec une version console, **JavaExe_console.exe**, permettant d'avoir la console DOS pour d'éventuels sortie écrans. Tout ce qui sera dit sur **JavaExe.exe** s'applique à **JavaExe_console.exe**.

Exemple :

Si ma classe principale se nomme *MyApp.class*, je copie puis renomme **JavaExe.exe** en **MyApp.exe**

Si ma classe principale est contenue dans un .jar, celui-ci devra aussi s'appeler *MyApp.jar*.

Les .class ou .jar doivent se trouver dans le même répertoire que le .EXE ou dans un répertoire nommé par défaut « resource » à créer au même niveau que le .EXE. Toutefois ce répertoire peut être défini spécifiquement en modifiant la propriété « **ResourceDirectory** » (voir le paragraphe intitulé *Les Propriétés*).

Exemple :

Si **MyApp.exe** se trouve dans le répertoire "**D:\Dev**", alors *MyApp.class* ou *MyApp.jar* se trouvent :

- soit dans "**D:\Dev**"
- soit dans le répertoire "**D:\Dev\resource**"

JavaExe reste toutefois dépendant d'un JDK ou d'un JRE, il est nécessaire qu'au moins un Java Runtime Environment (JRE) soit installé. Si **JavaExe** ne détecte pas de JDK ou JRE, il ouvrira un browser sur le site de [Sun](#) pour télécharger le JRE courant.

Vous pouvez fournir un JRE avec votre application (le JRE totalement décompacté et non pas le fichier d'installation). Dans ce cas, vous devez le mettre dans un répertoire nommé « jre », lui-même dans le répertoire du .EXE ou dans le répertoire « resource ».

Exemple :

Soit la configuration suivante de **MyApp.exe** :

- le .exe se trouve dans "**D:\Dev**"
- un JRE est fourni avec l'application et se trouve dans le répertoire "**D:\Dev\resource\jre**"

Alors **MyApp.exe** se lancera toujours avec ce JRE là quelque soit celui installé sur la machine cliente, même s'il n'y en a aucun d'installé.

Les propriétés

Une fois le fichier exécutable créé, il est possible d'y associer des propriétés pour définir la manière dont l'application Java sera lancée ou pour spécifier certains paramètres nécessaires à son fonctionnement.

Ces propriétés sont à mettre dans un fichier texte portant le même nom que le fichier exécutable, mais avec l'extension « .properties ». Une propriété sera défini par un nom suivi de sa valeur, de la forme : « *nom = valeur* ». Toutefois ce fichier pourra être intégré au .exe en utilisant l'utilitaire **UpdateRsrcJavaExe**.

Exemple :

Si **MyApp.exe** se trouve dans le répertoire "**D:\Dev**", alors **MyApp.properties** peut se trouver dans ce même répertoire ou dans "**D:\Dev\resource**".

Dans cet exemple, **MyApp.properties** contient :

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

MyApp sera alors lancé avec Java 1.2 (ou plus), et la commande en ligne correspondante est :
java -classpath .;.\resource\MyApp.jar;.\resource\classes12.zip MyApp "test" 123

Voici la liste de ces propriétés :

- **JRE version** (ou **JREversion**) = pour spécifier la version minimum de java : **1.4** ; **1.3** ; ...
Si un JRE est fourni avec l'application, cette propriété sera ignorée.
exemple :
JREversion = 1.3 JavaExe doit pouvoir trouver au moins la version 1.3 de Java pour lancer l'application
- **Run Type** (ou **RunType**) = pour spécifier comment l'application doit être lancé :
0 = en tant que simple application (valeur par défaut)
1 = en tant que service
2 = en tant que Panneau de configuration (ControlPanel)
exemple :
RunType = 1 JavaExe lancera l'application en tant que Service
- **Run As Service** (ou **RunAsService**) = cette propriété ne devrait plus être utilisé. A remplacer par « **RunType = 0** » ou « **RunType = 1** »
- **Main Class** (ou **MainClass**) = pour indiquer le nom complet de votre classe principale, dans le cas où JavaExe ne pourrait pas la trouver d'après seulement le nom de l'exécutable ou du Manifest dans le .jar. Le seul cas où il est nécessaire de spécifier cette propriété sera lorsque le nom du .exe et du .jar ne reflète pas le nom de la classe principale et aucun Manifest n'est trouvé.
exemple :
MainClass = com.toto.myClass
- **Main Args** (ou **MainArgs**) = ces valeurs seront passées en arguments à la méthode main de votre classe principale, dans la variable (String[] args).
exemple :
MainArgs = 123 aze l'argument args[] de la méthode main contiendra : [0] = « 123 » et [1] =

« aze ».

- **Personal Options** (ou **PersonalOptions**) = permet de spécifier les options de lancement propres à la JVM.

exemple :

PersonalOptions = -Xms64m -Xverify:none

- **Personal Classpath** (ou **PersonalClasspath**) = si votre application a besoin de .jar, .zip ou .class supplémentaires ou se trouvant dans d'autres répertoires. Plusieurs fichiers ou répertoires peuvent être spécifiés en les séparant d'un point-virgule.

exemple :

PersonalClasspath = D:\Dev\lib\lib.jar ; C:\Application\resource

- **Resource Directory** (ou **ResourceDirectory**) = pour indiquer le répertoire ressource contenant les JAR, les DLL, les images, les fichiers de propriétés,... Si ce paramètre est absent, le répertoire nommé « resource » situé au même niveau que le .EXE sera utilisé par défaut.

exemple :

ResourceDirectory = .\bin spécifie ce répertoire où les .jar principaux doivent être recherchés par défaut.

- **Class Directory** (ou **ClassDirectory**) = pour indiquer le ou les répertoires (séparés par ';') à scanner récursivement afin d'y trouver tous les .jar et .zip à mettre dans le ClassPath. Cette propriété contiendra d'office au moins le répertoire « resource » permettant ainsi la prise en compte de tous les .jar contenu dans ce répertoire sans devoir les spécifier un par un dans le classpath.

exemple :

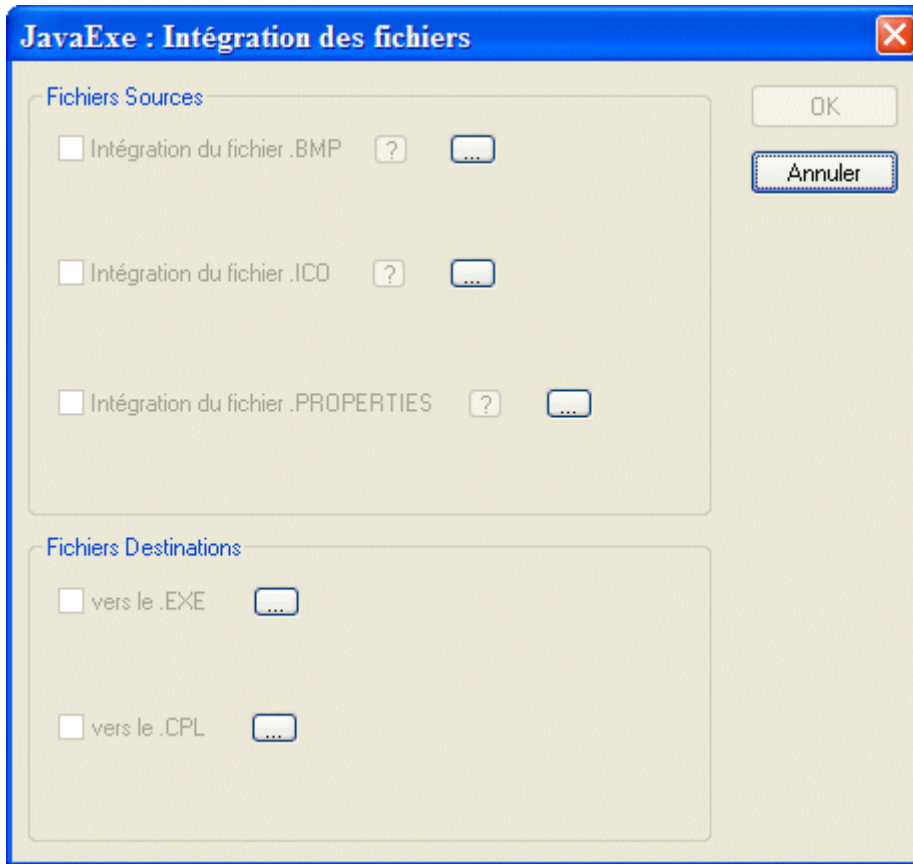
ClassDirectory = .\lib\ ; D:\Dev\lib ajoute à ClassPath tous les .jar et .zip trouvés dans ces 2 répertoires et leurs sous-répertoires respectifs, ainsi que dans le répertoire « resource ».

- **Path JRE** (ou **PathJRE**) = chemin du JRE s'il est fourni avec l'application. Par défaut il sera recherché dans le répertoire « jre » au même niveau que le .exe ou dans le répertoire « resource ».
- **Path Browser** (ou **PathBrowser**) = chemin du browser à utiliser pour l'installation éventuel d'un JRE (par défaut c'est le chemin d'Internet Explorer).
- **Display BoxInstall** (ou **Display_BoxInstall**) = pour indiquer si un message doit être affiché lorsque JavaExe ne trouve pas de JRE ou JDK, et demandant si l'on désire installer un JRE ou quitter l'application. Seulement deux valeurs sont acceptées : 0 ou 1.
1 = affiche la boîte de dialogue pour installer ou non le JRE (valeur par défaut)
0 = n'affiche aucun message, et entame la procédure d'installation en ouvrant un browser sur l'URL adéquate.
- **URL InstallJRE** (ou **URL_InstallJRE**) = permet d'indiquer une URL sur laquelle JavaExe ouvrira un browser dans le cas où aucun JRE ou JDK ne sera trouvé au lancement de l'application. Si cette propriété n'est pas indiquée, c'est l'URL sur java.sun.com qui sera pris en compte.


Il peut y avoir d'autres propriétés si votre application utilise ce même fichier pour ses besoins propres.

Utilisation de UpdateRsrcJavaExe (ex MergeICO)

JavaExe est fourni avec un autre programme, **UpdateRsrcJavaExe** (appelé MergeICO dans les versions précédentes), permettant de changer l'icône de votre **MyApp.exe**, de définir un écran de démarrage (le Screen Splash), ou encore d'intégrer le fichier des propriétés dans le .exe ou dans le .cpl (pour l'utilisation de l'application Java en tant que ControlPanel).






L'intégration de ces fichiers peut se faire de trois façons :

- En cliquant sur le bouton  du type de fichier que l'on souhaite ouvrir.
- En déplaçant les fichiers voulus sur cette fenêtre d'**UpdateRsrcJavaExe**.
- En ligne de commande.

Les types de fichiers pris en compte :

- **.BMP** : permet de définir un écran de démarrage à l'application Java.
- **.ICO** : permet de changer l'icône du fichier .exe
- **.PROPERTIES** : permet d'intégrer les propriétés utilisées par JavaExe.
- **.EXE** : permet de spécifier le .exe dérivé de JavaExe.exe (renommé ou pas) qui recevra les fichiers à intégrer.
- **.CPL** : permet de spécifier le .cpl dérivé de JavaExe.cpl (renommé ou pas) qui recevra les fichiers à intégrer (seulement le fichier de propriétés peut être intégré à un .cpl).

Après avoir chargé un fichier à intégrer, il est possible d'en voir les caractéristiques en cliquant sur son bouton  .

Lorsqu'au moins un fichier source et un fichier destination seront chargés dans **UpdateRsrcJavaExe**, il sera alors possible de cliquer sur le bouton  pour exécuter l'intégration des fichiers dont la case  sera cochée.

Si **UpdateRsrcJavaExe** est utilisé en ligne de commande, voici la liste des arguments reconnus :

- **-run** : permet de lancer l'intégration sans que la fenêtre ai besoin de s'ouvrir si tous les paramètres nécessaires sont renseignés.
- **-exe=fichier** : pour indiquer le nom d'un fichier .exe qui recevra les fichiers à intégrer. Ce fichier exécutable doit être un dérivé de JavaExe.exe.
- **-cpl=fichier** : pour indiquer le nom d'un fichier .cpl qui recevra les fichiers à intégrer. Ce fichier doit être un dérivé de JavaExe.cpl.
- **-ico=fichier** : permet d'indiquer le nom d'une icône qui sera intégrée au .exe.
- **-bmp=fichier** : permet d'indiquer le nom d'une image au format BMP qui sera intégrée au .exe et servant d'écran de démarrage.
- **-prp=fichier** : pour spécifier le nom d'un fichier .properties qui sera intégré au .exe et au .cpl.

Changer l'icône du .EXE

Il est possible de modifier l'icône du fichier exécutable pour lancer votre application Java. Tous les formats d'icônes sont maintenant acceptés par **JavaExe**.

Pour ce faire il suffit d'utiliser **UpdateRsrcJavaExe** fourni avec **JavaExe**. (cf le paragraphe précédent pour son utilisation).

Ecran de démarrage (le Screen Splash)

Pour définir un écran de démarrage à votre application Java il suffit d'avoir l'image au format .BMP et d'utiliser le programme **UpdateRsrcJavaExe**. (cf le paragraphe traitant de cet utilitaire).

Lancement en tant qu'application

Pour lancer votre programme Java en tant qu'application Windows, vous n'avez rien de spécial à faire si ce n'est ce qui a déjà été dit dans le chapitre « Utilisation générale ».

Si l'application Java est lancée avec un écran de démarrage, il est possible d'en contrôler le temps d'apparition. Pour cela il suffit de définir la méthode statique suivante dans la classe principale :

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

Tant que cette méthode renvoie **FALSE**, l'écran de démarrage restera visible. Si elle renvoie **TRUE**, l'écran disparaît et la méthode ne sera plus appelée. Si cette méthode n'est pas définie, l'écran de démarrage restera un certain temps, fixé par **JavaExe**.

Il est également possible de contrôler le nombre d'instance de l'application Java, en autorisant ou pas un seul exemplaire en cours d'exécution. Pour cela votre classe principale doit contenir une méthode statique nommée « **isOneInstance** » et devra avoir la signature suivante :

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

Les arguments envoyés à cette méthode sont ceux qui seront envoyé à la méthode *main*. Si *isOneInstance* retourne **TRUE** alors une seule instance de l'application sera lancée.

Lors du lancement de l'application, si c'est la première instance en cours d'exécution, cette méthode ne sera pas appelée mais la méthode *main* avec ses éventuels arguments.

En revanche, si ce n'est pas la première exécution, la méthode *isOneInstance* de la première instance de l'application sera d'abord appelée avec les arguments que la méthode *main* aurait reçus. Si *isOneInstance* renvoie **TRUE** le processus s'arrête là et l'instance en cours de lancement sera annulée. Si *isOneInstance* renvoie **FALSE** le processus de lancement continue, une nouvelle instance de l'application sera exécutée et sa méthode *main* sera appelée avec les éventuels arguments.

Lancement en tant que service

Pour que votre application Java soit lancée en tant que service système, il suffit de créer le .exe (voir le chapitre « Utilisation générale ») et de spécifier dans le fichier .properties, la propriété « RunType = 1 ».

Il faut toutefois noter une restriction : le service ne pourra pas se lancer en version console avec **JavaExe_console**.

Au lancement de l'application plusieurs cas de figure peuvent se présenter :

1. la classe principale est prévue pour fonctionner comme une application normale, c'est-à-dire que le point d'entrée est `main()`.
2. l'application Java contient les méthodes définies pour **JavaExe** servant d'interface entre la gestion du service Windows et l'application (cf. plus bas, ainsi qu'en Annexe l'interface **JavaExe_I_ServiceManagement**).

Et pour chacun de ces cas, l'application-service peut être lancée directement avec la JVM ou via la commande **java.exe**. Cela nous fait donc 4 cas de lancement à étudier.

1. **main() + JVM** => le point d'entrée étant `main()`, celui-ci ne sera appelé que pour lancer le service, et ce dernier ne pourra être stoppé qu'en redémarrant le système.
2. **main() + java.exe** => idem que précédemment.
3. **interface + JVM** => les méthodes définies pour servir d'interface seront appelées individuellement selon les besoins. La méthode `main()` ne sera jamais appelée.
4. **interface + java.exe** => puisque le lancement s'effectue avec **java.exe**, le point d'entrée sera alors `main()` et nous retombons dans la configuration du cas n° 2.

Dans le cas n° 3, si pour une quelconque raison on ne peut pas appeler directement la JVM, on devra passer par **java.exe** (cas n°4) et donc la méthode `main()` sera le seul point d'entrée. Aussi, il est important de ne pas oublier d'appeler la méthode `serviceInit()` depuis `main()`. Pour plus de détails voir l'exemple fourni avec cette documentation.

Il est possible de lancer directement des opérations sur le service, comme son installation, sa suppression, son démarrage ou son arrêt, sans passer par les éventuels boîtes de dialogue de confirmation.

Pour cela il suffit de lancer **JavaExe.exe** (c'est-à-dire **MyApp.exe**) avec comme argument :

-installService	: pour forcer son installation
-deleteService	: pour forcer sa suppression
-startService	: pour forcer son démarrage
-stopService	: pour forcer son arrêt

Méthodes servant d'interface : JavaExe_I_ServiceManagement

Ces méthodes sont directement appelées par **JavaExe** :

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

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

```

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

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

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

```

Ces méthodes sont à déclarer soit dans la classe principale, soit dans une classe de même nom mais post-fixée par « _ServiceManagement ». Par exemple, si ma classe principale s'appelle *MyApp*, alors ces méthodes peuvent se trouver indifféremment dans *MyApp.class* ou dans *MyApp_ServiceManagement.class*. Il n'est pas nécessaire de toutes les déclarer.

1. **serviceIsCreate** : Cette méthode est appelée au lancement de *JavaExe.exe* (c'est-à-dire *MyApp.exe*) si le service n'est pas encore installé. Le service sera installé seulement si cette méthode renvoie **TRUE**. Si cette méthode n'est pas déclarée, une boîte de dialogue Windows s'ouvrira pour demander à l'utilisateur s'il souhaite ou non installer le service. La méthode **serviceGetInfo** sera également appelée pour obtenir certaines caractéristiques nécessaires à la création du service.
2. **serviceIsLaunch** : Cette méthode est appelée après l'installation du service. Celui-ci sera immédiatement lancé si la méthode renvoie **TRUE**. Une boîte de dialogue Windows s'ouvrira, si cette méthode n'est pas déclarée, pour demander à l'utilisateur s'il souhaite ou non lancer le service.
3. **serviceIsDelete** : Cette méthode sera appelée au lancement de *JavaExe.exe* (c'est-à-dire *MyApp.exe*) si le service est déjà installé. Le service sera supprimé seulement si cette méthode renvoie **TRUE**. Si cette méthode n'est pas déclarée, une boîte de dialogue Windows s'ouvrira pour demander si l'utilisateur souhaite ou non supprimer le service. Toutefois si le service a été créé en spécifiant que son arrêt n'était pas autorisé (cf. la méthode **serviceGetInfo**), le service ne sera effectivement supprimé qu'au redémarrage du système.
4. **serviceInit** : Cette méthode est appelée lorsque le service est lancé, que ce soit manuellement ou automatiquement. La méthode doit renvoyer **TRUE** si et seulement si l'application est active et en cours d'exécution. Si elle renvoie **FALSE** ou si elle ne répond pas avant un délai de 30 secondes, Windows considérera que le service a échoué à la tentative de démarrage et lancera alors le programme en cas d'échec s'il a été défini (cf. la méthode **serviceGetInfo**). Si la méthode n'est pas déclarée, le service sera lancé immédiatement sans condition.
5. **serviceFinish** : Cette méthode sera appelée lorsque le service aura été arrêté soit manuellement, soit automatiquement avec l'arrêt du système.
6. **serviceGetInfo** : Cette méthode est appelée au moment de la création du service afin d'obtenir certaines informations complémentaires, tels que :
 - Nom complet du service par opposition au nom court qui est le nom de l'exécutable.
 - Description du service.
 - « 1 » ou « **TRUE** » pour indiquer que le service sera lancé automatiquement avec le système.
 - « 1 » ou « **TRUE** » pour indiquer que le service peut être arrêté manuellement.
 - Nom du fichier à exécuter lorsque le service a échoué. Les fichiers .BAT peuvent ne pas s'exécuter correctement sur Windows 2000.
 - Arguments à fournir au programme qui s'exécute lors d'un échec.
 - Liste de noms de services, séparés par une tabulation ('\t'), dont ce service dépend. C'est-à-dire que Windows s'assurera que ces services sont lancés avant de lancer celui-ci.

Cette méthode renvoie un tableau de **string** dont les éléments correspondent respectivement à ceux cités précédemment. Si cette méthode n'est pas définie, toutes ces informations seront vides, le lancement sera automatique et l'arrêt ne sera pas autorisé. Cette méthode peut être appelée plusieurs fois par **JavaExe**.

7. **serviceControl_Pause** : Cette méthode est appelé lorsque Windows tente de mettre en pause le service. Celui-ci sera effectivement en pause si la méthode renvoie **TRUE** avant un délai de 30 secondes. Si la méthode n'est pas déclarée, le service sera mis en pause immédiatement.
8. **serviceControl_Continue** : Cette méthode est appelé lorsque Windows tente de relancer le service mise en pause. Celui-ci sera effectivement actif si la méthode renvoie **TRUE** avant un délai de 30 secondes. Si la méthode n'est pas déclarée, le service sera relancé immédiatement.
9. **serviceControl_Stop** : Cette méthode est appelé lorsque Windows tente de stopper le service. Celui-ci sera effectivement stoppé si la méthode renvoie **TRUE** avant un délai de 30 secondes. Après l'arrêt du service, la méthode **serviceFinish** sera finalement appelée. Si la méthode n'est pas déclarée, le service sera arrêté immédiatement.
10. **serviceControl_Shutdown** : Cette méthode est appelé lorsque Windows est arrêté ou redémarré. Elle a le même comportement que **serviceControl_Stop**.

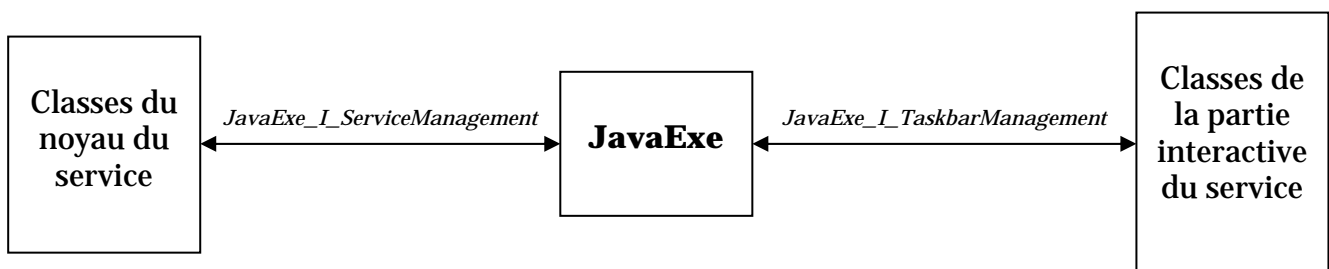
Les 3 méthodes suivantes sont utilisées pour les services qui interagissent avec le Bureau. Pour qu'un service soit reconnu comme Interactif, il suffit que votre application Java intègre la gestion de la barre de tâche (cf le chapitre « Gestion de la barre des tâches »). Le service ne peut pas communiquer directement avec le Bureau, il devra passer par des méthodes prévues à cet effet :

11. **serviceDataFromUI** (*Serializable* data) : Cette méthode sera appelée par la partie interactive du service avec en argument un objet à traiter par le service.
12. **serviceIsDataForUI** : Cette méthode devra renvoyer **TRUE** si un objet est disponible pour la partie interactive.
13. **serviceDataForUI** : Cette méthode renvoie un objet pour la partie interactive.

A ces trois méthodes correspond leur contrepartie dans le TaskbarManagement. Voir le chapitre « Gestion de la barre des tâches » pour le détail de ces méthodes, ainsi que l'interface **JavaExe_I_TaskbarManagement** en Annexe.

Il est important de comprendre qu'il ne doit pas y avoir de lien direct entre les classes du service proprement dit et les classes de sa partie interactive avec le Bureau. Si cela devait toutefois arriver, il s'agira de deux instances différentes de la même classe et donc avec des données différentes.

Voici un schéma résumant la structure d'un service interactif :



Bien entendu, du point de vue du développeur Java tout ceci est transparent. Il devra simplement veiller à ce que ses classes de la partie interactive ne fasse pas référence aux classes de la partie noyau, et vice-versa.

Lancement en tant que Panneau de contrôle

Pour que votre application Java soit reconnue comme un Panneau de contrôle, il suffit de créer le .exe (voir le chapitre « Utilisation générale ») et de spécifier dans le fichier .properties, la propriété « RunType = 2 ».

JavaExe est fourni aussi avec un autre type de fichier, le **JavaExe.cpl**, qui devra être renommé comme pour le .exe. C'est ce fichier qui sera reconnu comme un panneau de contrôle par Windows.

Il est possible de lancer directement son installation, ou sa suppression, sans passer par les éventuels boîtes de dialogue de confirmation.

Pour cela il suffit de lancer **JavaExe.exe** (c'est-à-dire **MyApp.exe**) avec comme argument :

-installCPL : pour forcer son installation
-deleteCPL : pour forcer sa suppression

Méthodes servant d'interface : *JavaExe_I_ControlPanelManagement*

Ces méthodes sont directement appelées par **JavaExe** :

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

Ces méthodes sont à déclarer soit dans la classe principale, soit dans une classe de même nom mais post-fixée par « *_ControlPanelManagement* ». Par exemple, si ma classe principale s'appelle *MyApp*, alors ces méthodes peuvent se trouver indifféremment dans *MyApp.class* ou dans *MyApp_ControlPanelManagement.class*. Il n'est pas nécessaire de toutes les déclarer.

1. **cplIsCreate** : Cette méthode est appelée au lancement de **JavaExe.exe** (c'est-à-dire **MyApp.exe**) si le panneau de contrôle n'est pas encore installé. Il sera installé seulement si cette méthode renvoie **TRUE**. Si cette méthode n'est pas déclarée, une boîte de dialogue Windows s'ouvrira pour demander à l'utilisateur s'il souhaite ou non installer le panneau de contrôle. La méthode **cplGetInfo** sera également appelée pour obtenir certaines caractéristiques nécessaires à la création du panneau de contrôle.
2. **cplIsDelete** : Cette méthode sera appelée au lancement de **JavaExe.exe** (c'est-à-dire **MyApp.exe**) si le panneau de contrôle est déjà installé. Il sera supprimé seulement si cette méthode renvoie **TRUE**. Si cette méthode n'est pas déclarée, une boîte de dialogue Windows s'ouvrira pour demander si l'utilisateur souhaite ou non supprimer le panneau de contrôle.
3. **cplGetInfo** : Cette méthode est appelée au moment de la création du panneau de contrôle afin d'obtenir certaines informations complémentaires, tels que :
 - Nom.
 - Description.
 - Ses catégories d'appartenance (à partir de la version XP de Windows). Si vous voulez faire figurer votre panneau de contrôle dans plusieurs catégories, vous devrez séparer chaque valeur par une

virgule (','). Voir en Annexe l'interface « *JavaExe_I_ControlPanelManagement* » pour la liste des ces catégories.

Cette méthode renvoie un tableau de **string** dont les éléments correspondent respectivement à ceux cités précédemment. Si cette méthode n'est pas définie, toutes ces informations seront vides et le nom sera celui de l'exécutable.

4. **cplOpen** : Cette méthode sera appelée à l'ouverture du panneau de contrôle. Si cette méthode n'est pas déclarée, c'est la méthode *main* qui sera appelé.

Fonctionnalités additionnelles

Gestion des Événements Système

Cette fonctionnalité de **JavaExe** permet à l'application Java de recevoir certain événement de Windows, telle qu'une connexion ou déconnexion à un réseau, un changement d'affichage, un début ou fin d'une session, etc.

Pour ce faire, il doit exister une méthode qui servira d'interface entre **JavaExe** et votre application Java, dont la signature est de la forme :

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

Cette méthode est à déclarer soit dans la classe principale, soit dans une classe de même nom mais post-fixée par « `_SystemEventManager` ». Par exemple, si ma classe principale s'appelle *MyApp*, alors cette méthode peut se trouver indifféremment dans *MyApp.class* ou dans *MyApp_SystemEventManager.class*.

La même méthode est utilisée pour tous les types d'événements et ses arguments dépendent du message reçu. La valeur retournée par **notifyEvent** dépend aussi du message.

Le premier argument, *msg*, contient le type d'événement et en voici la liste des différentes valeurs :

- **WM_COMPACTING** : Ce message est reçu lorsque le système commence à saturer.
- **WM_CONSOLE** : Ce message est envoyé lorsque **JavaExe** est lancé en mode console (*JavaExe_console.exe*) et qu'une tentative d'interruption à eu lieu. L'argument *val1* contient le type d'interruption :
 - **CTRL_C_EVENT** : un CTRL-C est déclenché, mais sera annulé si **notifyEvent** renvoie 0.
 - **CTRL_BREAK_EVENT** : est utilisé par Java pour le dump des threads actifs, mais sera annulé si **notifyEvent** renvoie 0.
 - **CTRL_CLOSE_EVENT** : l'utilisateur tente de fermer la fenêtre DOS et cette tentative sera annulée si **notifyEvent** renvoie 0.
 - **CTRL_LOGOFF_EVENT** : l'utilisateur à déclenché la fermeture de sa session. Quelque soit la valeur retourné par **notifyEvent** cette fermeture ne sera pas interrompue.
 - **CTRL_SHUTDOWN_EVENT** : l'utilisateur à déclenché l'arrêt du système. Quelque soit la valeur retourné par **notifyEvent** le système sera arrêté.
- **WM_DEVICECHANGE** : Ce message signifie qu'une modification hardware a eu lieue ou nécessite une confirmation. Par exemple si un périphérique a été inséré ou retiré, ou un CD-ROM, etc. Les arguments utilisés sont :
 - *val1* : la nature de la modification :
 - DBT_QUERYCHANGECONFIG**
 - DBT_CONFIGCHANGED**
 - DBT_CONFIGCHANGECANCELED**
 - DBT_DEVICEARRIVAL**
 - DBT_DEVICEQUERYREMOVE**
 - DBT_DEVICEQUERYREMOVEFAILED**
 - DBT_DEVICEREMOVECOMPLETE**

DBT_DEVICEREMOVEPENDING DBT_DEVICETYPESPECIFIC

- **val3** : nom du port. Utilisé seulement par **DBT_DEVTYP_PORT**.
- **arr1** : tableau de au plus 5 **int** (cela dépend de **val1**)
 - [0]**
 - [1]** = type du device
 - DBT_DEVTYP_OEM**
 - DBT_DEVTYP_VOLUME**
 - DBT_DEVTYP_PORT**
 - [2]**
 - [3]** = si **[1]=DBT_DEVTYP_VOLUME =>** valeur où chaque position binaire correspond à un n° de lecteur : bit 0 = lecteur A ; bit 1 = lecteur B ; ... ; bit 26 = lecteur Z.
 - [4]** = si **[1]=DBT_DEVTYP_VOLUME =>** 1=Lecteur Multimédia (CD-ROM, ...) ; 2=Lecteur réseau
- **WM_DISPLAYCHANGE** : Cet événement est reçu lorsque la résolution de l'écran a changé. Les arguments utilisés sont :
 - **val1** : le nombre de bits par pixel. On en déduit le nombre de couleurs par 2^{val1} .
 - **val2** : une valeur sur 32 bits décomposée comme suit : 31...16 15...0. Les bits de 0 à 15 correspondent à la largeur de l'écran. Les bits de 16 à 31 donnent la hauteur. Pour dissocier ces 2 valeurs, il suffit d'appliquer :
 - $w = (val2 \& 0x0000FFFF);$
 - $h = ((val2 \gg 16) \& 0x0000FFFF);$
- **WM_ENDSESSION** : Ce message est reçu lorsque la session de l'utilisateur va être fermée. Soit parce que l'utilisateur se déconnecte de son login, soit que le système est arrêté. Ce message n'est pas reçu si l'application Java est lancé en mode console. Les arguments utilisés sont :
 - **val1** : contient la valeur 1 si la fermeture de la session a été confirmée, sinon 0. (voir le message **WM_QUERYENDSESSION** pour cette confirmation).
 - **val2** : permet de savoir s'il s'agit d'une simple déconnection de l'utilisateur ou de l'arrêt du système. Si cet argument contient la valeur **ENDSESSION_LOGOFF** alors il s'agit d'une déconnection. Il est préférable de tester la présence de cette valeur (en tant que bits) plutôt que la stricte égalité :
 - $((val2 \& \mathbf{ENDSESSION_LOGOFF}) \neq 0)$ est préférable à
 - $(val2 == \mathbf{ENDSESSION_LOGOFF})$
- **WM_NETWORK** : Cet événement est reçu lorsque l'état du réseau a changé. Les arguments utilisés sont :
 - **val1** : le type de changement
 - NET_DISCONNECT**
 - NET_CONNECTING**
 - NET_CONNECTED**

- **val3** : le nom de l'interface réseau concerné.
- **arr1** : un tableau de 13 **int** utilisé comme suit :
 - [0]** = type du réseau
 - 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]** = les 4 champs de l'IP cliente.
 - [5...8]** = les 4 champs de l'IP de la gateway.
 - [9...12]** = les 4 champs du masque réseau.
- **WM_POWERBROADCAST** : Cet événement est déclenché lorsque l'état de la batterie ou de l'alimentation a changé. Les arguments utilisés sont :
 - **val1** : le type d'état
 - PBT_APMQUERYSPEND**
 - PBT_APMQUERYSPENDFAILED**
 - PBT_APMSPEND**
 - PBT_APMRESUMECRITICAL**
 - PBT_APMRESUMESPEND**
 - PBT_APMBATTERYLOW**
 - PBT_APMPOWERSTATUSCHANGE**
 - PBT_APMOEMEVENT**
 - PBT_APMRESUMEAUTOMATIC**
 - **val2** : autorise ou non une interaction avec l'utilisateur (comme afficher une boîte de dialogue, ...). Si cet argument contient 0 aucune interaction ne sera autorisée.
 - **arr1** : contient 2 **int**
 - [0]** = nombre de secondes actuellement utilisable en batterie.
 - [1]** = nombre total de secondes utilisable en batterie (capacité maximale de la batterie).
 - **arr2** : contient 3 **byte**
 - [0]** = 1 si la batterie est sur le secteur, sinon 0.
 - [1]** = état de chargement de la batterie (ou 255 si inconnu).
 - [2]** = pourcentage de chargement (ou 255 si inconnu).
- **WM_QUERYENDSESSION** : Cet événement est déclenché lorsque la session va être interrompue. Une confirmation est d'abord demandée à l'utilisateur et si **notifyEvent** renvoie 0 la session ne sera pas interrompue. Un autre message, **WM_ENDSESSION**, sera automatiquement envoyé dans tous les cas, après celui-ci avec le résultat de **notifyEvent**. Ce message n'est pas reçu si l'application Java est lancée en mode console. Les arguments utilisés sont :
 - **val2** : même signification que pour le message **WM_ENDSESSION**.

- **WM_SESSION_CHANGE** : Ce message est reçu lorsque qu'un utilisateur se connecte, déconnecte ou verrouille la session. Les arguments utilisés sont :
 - **val1** : contient la raison qui a déclenché cet événement
 - 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** : contient le numéro de la session concerné, dans le cas où plusieurs sessions peuvent être actives en même temps.
 - **val3** : contient le nom du domaine et celui de l'utilisateur (son login) qui est à l'origine de l'événement. Cette information est de la forme *domaine\login*.
 - **arr1** : est utilisé seulement par **WTS_SESSION_LOGGED** et contient un seul élément indiquant si l'utilisateur connecté est celui qui est actuellement actif.
- **WM_SYSCOMMAND** : Cet événement regroupe divers autres événements.
 - **val1** : le type de l'événement (pour l'instant seul **SC_SCREENSAVE** est pris en compte)
 - **val2** : état de l'économiseur d'écran : **1** pour le démarrage, **0** pour l'arrêt.
- **WM_TIMECHANGE** : Cet événement a lieu lorsque l'heure du système a changé. Aucun argument n'est utilisé.
- **WM_USERCHANGED**

Voir en Annexes l'interface *JavaExe_I_SystemEventManager* pour la valeur des constantes utilisées, et aussi les exemples 6 et 8 traitants des événements systèmes.

Gestion de la barre des tâches

Cette fonctionnalité permet à l'application Java d'avoir son icône dans la barre des tâches ainsi qu'éventuellement un ou deux menus associés (un menu pour le click droit et un autre pour le click gauche).

Méthodes servant d'interface : *JavaExe_I_TaskbarManagement*

Ces méthodes sont directement appelées par **JavaExe** :

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

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

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

public static 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 ();
```

Ces méthodes sont à déclarer soit dans la classe principale, soit dans une classe de même nom mais post-fixée par « *_TaskbarManagement* ». Par exemple, si ma classe principale s'appelle *MyApp*, alors ces méthodes peuvent se trouver indifféremment dans *MyApp.class* ou dans *MyApp_TaskbarManagement.class*. Il n'est pas nécessaire de toutes les déclarer.

1. **taskGetMenu** (*boolean* isRightClick, *int* menuID) : Cette méthode est appelée pour obtenir la liste des entrées du menu associé à l'icône dans la barre des tâches. Ce menu sera géré par Windows, toutefois si l'application Java possède et gère elle-même son propre menu pour l'icône, cette méthode ainsi que **taskGetDefaultMenuID** et **taskDoAction** seront inutiles (cf. la méthode **taskDisplayMenu** pour les menus propres à l'application). Dans ce cas il ne sera pas nécessaire de la déclarer, ou alors elle devra renvoyer la valeur `null`.

Cette méthode possède 2 arguments :

- **isRightClick** : contient `TRUE` si le menu à afficher correspond à un click droit de la souris. Il peut donc y avoir 2 menus différents selon qu'il s'agisse du click droit ou gauche.
- **menuID** : dans le cas où la liste des entrées à renvoyer correspond à celle d'un sous-menu, cet argument contient le numéro de l'entrée possédant ce sous-menu. Sinon la valeur de l'argument est à `0` (ou négatif).

La liste renvoyée par cette méthode est de type *String*[[[]], c'est-à-dire une liste qui contient une liste de valeurs de types *String*. A chaque entrée du menu correspond une liste de valeurs de la forme :

{ *ID*, *LABEL*, *TYPE*, *STATUS* }

Où :

- *ID* = un numéro unique pour cette entrée. La valeur doit être strictement supérieure à 0.
- *LABEL* = le texte qui sera affiché.
- *TYPE* = la nature de l'entrée (cf. plus bas).
- *STATUS* = l'état de l'entrée à l'affichage (cf. plus bas).

Avec *TYPE* :

MFT_MENUBARBREAK = place l'entrée dans une nouvelle colonne avec une séparation verticale.

MFT_MENUBREAK = place l'entrée dans une nouvelle colonne sans séparation.

MFT_RADIOCHECK = si l'entrée est dans l'état **checked**, elle sera alors affichée sous forme de radio-bouton.

MFT_SEPARATOR = affiche une séparation horizontale. *LABEL* est alors ignoré.

MFT_RIGHTORDER = affiche le texte de la droite vers la gauche.

Si aucune valeur n'est spécifiée, ce sera alors un texte simple, aligné de gauche à droite, qui sera affiché.

Les valeurs de *TYPE* sont mixables entre elles sauf **MFT_MENUBARBREAK** avec **MFT_MENUBREAK**. Par exemple :

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

Avec *STATUS* :

MFS_DISABLED = si l'entrée du menu est désactivée.

MFS_CHECKED = si l'entrée est checkée.

MFS_HILITE = si l'entrée est présélectionnée.

Si aucune valeur n'est spécifiée, l'entrée du menu sera alors simplement active et non-checkée.

Les valeurs de *STATUS* sont aussi mixables entre elles.

2. **taskGetDefaultMenuID** (*boolean* isRightClick) : Cette méthode permet de définir quelle est l'entrée du menu qui sera pris en compte lors du double-click sur l'icône. Cette entrée sera alors mise en gras dans le menu. Si cette méthode n'est pas déclarée ou si elle renvoie 0 (ou une valeur négative), aucune entrée ne sera définie.

L'argument de cette méthode, *isRightClick*, contient **TRUE** si cela concerne le menu pour le click droit de la souris.

3. **taskDoAction** (*boolean* isRightClick, *int* menuID) : Cette méthode prend en charge l'action à faire lorsqu'une entrée du menu aura été sélectionnée.

Cette méthode possède 2 arguments :

- *isRightClick* : contient **TRUE** si le menu concerné est celui du click droit de la souris.
- *menuID* : contient le numéro de l'entrée sélectionnée par l'utilisateur.

4. **taskDisplayMenu** (*boolean* isRightClick, *Component* parent, *int* x, *int* y) : Cette méthode prend en charge l'affichage et la gestion du menu associé à l'icône. Elle renvoie **TRUE** si le menu est pris en charge.

Cette méthode possède 4 arguments :

- **isRightClick** : contient **TRUE** si le menu à gérer correspond à un click droit de la souris. Il peut donc y avoir 2 menus différents selon qu'il s'agisse du click droit ou gauche.
- **parent** : selon la manière dont le menu sera géré par l'application Java, il peut être nécessaire d'avoir un objet parent auquel ce menu sera rattaché. Cet objet parent est créé par **JavaExe**.
- **x** et **y** : coordonnées où doit être affiché le menu, correspondant au coin inférieur droit de ce menu.

5. **taskGetInfo** : Cette méthode permet d'obtenir diverses informations pour l'affichage et la gestion de l'icône et de son menu. Cette méthode renvoie un tableau de **String** contenant dans l'ordre :

- La description de l'icône, qui sera affichée lorsque la souris passera dessus.
- Le type d'action à faire pour un simple click-droit de la souris (par défaut ce sera **ACT_CLICK_MENU**, cf. plus bas).
- Le type d'action à faire pour un double click-droit de la souris (**ACT_CLICK_NOP** par défaut).
- Le type d'action à faire pour un simple click-gauche de la souris (**ACT_CLICK_NOP** par défaut).
- Le type d'action à faire pour un double click-gauche de la souris (**ACT_CLICK_OPEN** par défaut).

Il y a 3 types d'action possible :

ACT_CLICK_NOP = ne rien faire
ACT_CLICK_OPEN = exécute l'action définie par la méthode **taskDoAction** avec l'entrée du menu renvoyée par la méthode **taskGetDefaultMenuID**.
ACT_CLICK_MENU = affiche le menu en appelant d'abord la méthode **taskDisplayMenu**. Si cette dernière n'est pas définie ou renvoie **FALSE**, alors la méthode **taskGetMenu** sera appelée.

6. **taskIsShow** : Cette méthode est régulièrement appelée par **JavaExe** pour savoir si l'icône doit être affichée ou cachée. Si la méthode renvoie **TRUE** l'icône sera affichée.
7. **taskInit** : Cette méthode est appelée au lancement de l'application.
8. **taskIsBalloonShow** : Cette méthode est régulièrement appelée par **JavaExe** pour savoir si un message est prêt à être affiché au niveau de l'icône. Si la méthode renvoie **TRUE**, alors la méthode **taskGetBalloonInfo** sera appelée pour obtenir le message.
9. **taskSetBalloonSupported** (**boolean** isSupported) : Cette méthode est appelée au lancement de l'application pour l'informer si la version de Windows supporte ou non les messages d'icône. Si l'argument de cette méthode contient **TRUE**, alors le système supporte ce type de message.
10. **taskGetBalloonInfo** : Cette méthode permet d'obtenir le message d'icône à afficher et quelques informations complémentaires. Elle sera appelée lorsque la méthode **taskIsBalloonShow** renverra **TRUE**, ainsi qu'au lancement de l'application. Cette méthode renvoie un tableau de **String** contenant dans l'ordre :
- Titre du message.
 - Message à afficher.
 - Type du message.
 - Durée d'affichage du message (en secondes).

Avec pour « Type du message » :

NIIF_NONE	= message neutre.
NIIF_INFO	= message d'information.
NIIF_WARNING	= message d'attention.
NIIF_ERROR	= message d'erreur.
NIIF_USER	= message propre à l'application.

11. **taskDataFromService** (*Serializable* data) : Cette méthode sera appelée par le service (dans le cas où l'application Java est lancée en mode service) avec en argument un objet à traiter par la partie interactive.
12. **taskIsDataForService** : Cette méthode devra renvoyer **TRUE** si un objet est disponible pour le service.
13. **taskDataForService** : Cette méthode renvoie un objet pour le service.

A ces trois méthodes correspond leur contrepartie dans le ServiceManagement. Voir le chapitre « Lancement en tant que service » pour le détail de ces méthodes et quelques explications complémentaires sur les services en interaction avec le Bureau, ainsi que l'interface **JavaExe_I_ServiceManagement** en Annexe.

Voir en Annexes l'interface **JavaExe_I_TaskbarManagement** pour la valeur des constantes utilisées, et aussi les exemples 4, 5 et 8 utilisant la gestion de la barre des tâches.

Annexes

Interfaces Java

Ces interfaces ne sont utilisées que pour avoir accès aux constantes nécessaires dans les différentes fonctionnalités de **JavaExe**. Les méthodes de classes qui y sont définies ne sont là qu'à titre indicatif puisque les méthodes définies dans une interface ne s'appliquent qu'à des instances.

ApplicationManagement

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

ControlPanelManagement

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

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

    public static String[] cplGetInfo ();

    public static void cplOpen ();
}
```

ServiceManagement

```
interface JavaExe_I_ServiceManagement
{
    public static boolean serviceIsCreate ();
}
```

```

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

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

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

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

```

SystemEventManager

```

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

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

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

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

    static final int ENDSESSION_LOGOFF            = 0x80000000;

    static final int SC_SCREENSAVE                 = 0xF140;

```

```

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

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

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

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

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

```

TaskbarManagement

```

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

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

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

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

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

```

```
public static void taskDoAction (boolean isRightClick, int menuID);  
public static boolean taskDisplayMenu (boolean isRightClick, Component parent, int x, int y);  
  
public static String[] taskGetInfo ();  
public static boolean taskIsShow ();  
public static void taskInit ();  
  
public static 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 ();  
}
```

Exemples

Ces exemples ne sont là que pour mettre en pratique les différentes fonctionnalités de **JavaExe**. Leur code source peut être utilisé comme point de départ pour des applications plus complexes.

1 – Application

Cet exemple montre simplement une application Java qui ouvre une boîte de dialogue. Il introduit la notion du .EXE pour lancer une application Java avec un écran de démarrage.

2 - Control Panel

Cet exemple définit un panneau de configuration Windows (un `ControlPanel`) contenant une boîte de dialogue à 3 onglets, où leurs valeurs seront lues depuis le fichier de propriétés associé à l'application (*Example2.properties*). En appuyant sur le bouton « Apply » ou « Ok » les valeurs seront sauvées dans ce même fichier.

Il y a 2 façons de lancer ce panneau de contrôle. Soit en double cliquant sur le .EXE pour installer/désinstaller le `ControlPanel`, soit en double cliquant sur le .CPL pour l'ouvrir directement.

3 – Service

Cet exemple définit un service sans interaction avec le Bureau. Lors de son installation, une boîte de dialogue propose différents paramètres pour le lancement du service, comme le n° de port. Ce service se met en attente de connexion sur le port ainsi défini et renvoie la date au client qui s'est connecté.

4 – TrayIcon

Cet exemple montre une application Java utilisant la gestion de la barre des tâches. L'icône de cette barre possède 2 menus, un pour le click droit et un pour le click gauche de la souris. En double cliquant sur l'icône, l'application ouvre une boîte de dialogue dans laquelle se trouve un checkbox pour afficher ou non l'icône de la barre des tâches.

5 - Service & TrayIcon

Cet exemple définit un service en interaction avec le Bureau. Ce service est le même que pour l'exemple 3. L'interaction avec le Bureau se passe par la barre des tâches dont l'icône possédera un menu sur le click droit de la souris. Depuis ce menu il sera possible de reconfigurer le service ou de lancer un browser sur le port d'écoute du service.

6 - System Event

Cet exemple, qui se lance comme une simple application, intercepte les événements systèmes de Windows et les affiche dans une boîte de dialogue. Si l'application est lancée en mode Dos, avec *Example6_console.exe*, le CTRL-C de la console sera aussi intercepté et une boîte de dialogue s'ouvrira pour en demander la confirmation.

7 – OneInstance

Cet exemple montre la fonctionnalité du « OneInstance » permettant de contrôler le nombre d'instance de l'application lancée en même temps. Lorsque l'application se lance, une boîte de dialogue s'ouvre contenant un checkbox. Lorsque celui-ci sera coché plusieurs instances de l'application pourront s'exécuter en même temps.

Si l'application est lancée avec des arguments, ceux-ci s'afficheront d'abord dans la boîte de dialogue de la 1^{ère} instance, ensuite selon que le checkbox est coché ou non ces mêmes arguments s'afficheront dans la boîte de dialogue de l'application fraîchement lancé.

8 - Service & TrayIcon & System Event

Cet exemple crée un service Windows en interaction avec le Bureau interceptant les événements systèmes. Il gère donc la barre des tâches dont l'icône possède un menu sur le click droit de la souris. A partir de ce menu il est possible d'ouvrir la boîte de dialogue affichant les événements reçu par le service ou de cacher les messages liés à cet icône.

JavaExe et **UpdateRsrcJavaExe** (ex-**MergeICO**) sont des créations et des propriétés de DevWizard (DevWizard@free.fr)

Vous êtes libre de vous en servir et de les fournir avec vos applications que vous souhaitez distribuer, qu'elles soient freeware, shareware ou commerciales.

Toutefois seul DevWizard est autorisé à apporter une quelconque modification à **JavaExe** et **UpdateRsrcJavaExe**.

Toute demande de modification peut être faite par mail à DevWizard@free.fr

© 2002-2006 by DevWizard