REFERENTIEL NORMATIF du CNES

RNC

Référence: RNC-CNES-Q-HB-80-527
Version 6
02 Juin 2008

MANUEL

ASSURANCE PRODUIT
REGLES POUR L'UTILISATION DU LANGAGE JAVA

ACCORD du Bureau de Normalisation	**BN n° 34 du 25/06/07 – BN n°44 du 08/09/08**
APPROBATION Président du CDN Alain CUQUEL	

PAGE D'ANALYSE DOCUMENTAIRE

TITRE : REGLES POUR L'UTILISATION DU LANGAGE JAVA	
MOTS CLES : Java , Applet, Application, Règle	
NORME EQUIVALENTE : Néant	
OBSERVATIONS : Néant	
RESUME : Ce document présente les règles de programmation applicables à tout projet SOL développé en JAVA.	
SITUATION DU DOCUMENT : Ce document appartient à la collection des Manuels du Référentiel Normatif du CNES (RNC). Il est affilié au document « RNC-ECSS-Q-ST-80 Software Product Assurance ».	
NOMBRE DE PAGES : 80	**LANGUE :** Française
Progiciels utilisés / version : Word 2002	
SERVICE GESTIONNAIRE : Inspection Générale Direction de la Fonction qualité (IGQ)	
AUTEUR(S) : **Repris par J-C. DAMERY**	**DATE : 02/06/2008**

PAGES DES MODIFICATIONS

VERSION	DATE	PAGES MODIFIEES	OBSERVATIONS
0	22/09/97	Création du document	Premier Draft élaboré par CAP GEMINI France (M-P. ETIENNE, O. LAGNEAU)
PR.1	14/12/97	Toutes	Version provisoire soumise à relecture
PR.2	18/05/98	Toutes	Intégration des remarques des relecteurs CNES
PR.3	15/01/99	Toutes	Reprise du document par VALTECH et introduction des remarques des relecteurs externes
1.0	03/03/99	Toutes	Validation et normalisation du document
2.PR	11/11/99	Toutes	Harmonisation avec la MPM " Démarche de développement objet pour les logiciels "
2.0	13/03/00	Toutes	Changement de codification du RNC
3	01/12/00	Toutes	Reprise du document par CRIL INGENIERIE et prise en compte de la version 1.3 du JDK
4	10/12/05	Entêtes et références	Modifications pour mise en conformité avec l'évolution de l'ECSS E40, et mise en conformité avec DDO.
5	10/12/2006	Toutes	Mise en commun de règles et mise en forme, avec le support de T. Leydier (Virtualité Réelle). Modification du titre. Cf. FEB 48/2006 acceptée au BN n° 22 du 06/03/06. Document accepté au BN n° 34 du 25/06/07 pour introduction dans le RNC.
6	02/06/2008	Toutes	Changement de nomenclature suite à la phase de benchmarking ECSS (ancienne référence RNC-CNES-Q-80-527).

TABLE DES MATIERES

1. INTRODUCTION

Le document « Règles pour l'utilisation du langage JAVA » est rattaché au document « RNC-ECSS-Q-ST-80 Software Product Assurance ». Il décrit les règles applicables pour un logiciel utilisant le langage JAVA.

Le langage Java est un langage de programmation, défini par Sun Microsystems, orienté objet et indépendant de la plate-forme matérielle et système.

En fonction du type d'application Java développée, certains manuels doivent être utilisées en complément, notamment en ce qui concerne la mise en œuvre de la méthode de développement et la prise en compte des contraintes de sécurité.

2. OBJET

Le but de ce document est de participer à la réutilisation du savoir-faire en établissant des règles d'utilisation du langage Java, afin de concentrer l'effort de réalisation sur l'application et sur le contenu. Ce document est indispensable pour toute utilisation du langage Java dans un projet du CNES.

Ce document énonce des règles applicables pour une application écrite en langage Java. Ces règles doivent être suffisamment précises pour être utilisables et efficaces, sans toutefois être trop restrictives, ce qui serait un obstacle à leur utilisation dans des contextes techniques qui peuvent être très différents les uns des autres.

2.1. LES DEUX TYPES D'UTILISATION DE JAVA

On peut attirer l'attention du lecteur sur les deux grandes catégories d'applications Java ; la distinction explicitée ci-dessous a des incidences sur le volume de l'application, les contraintes de sécurité et de portabilité, ainsi que sur les compétences nécessaires aux développeurs.

2.2. APPLET ET APPLICATION STANDALONE

Le langage Java doit sa popularité croissante à ses applications dans les techniques associées à Internet. Il permet en effet d'animer des applications clientes interactives sur le Web, le code de l'application étant téléchargé depuis un serveur lui-même connecté au Web, puis exécuté sur le client par la machine virtuelle Java.

Cette assimilation de Java à une technologie dédiée au Web tend à masquer le fait que ce langage est un langage orienté objet à part entière, plus simple et plus robuste que C++, portable, fourni avec de nombreuses bibliothèques réutilisables dont la définition fait partie du langage, et accompagné de puissants outils de documentation. Lorsqu'une application Java est activée depuis une page HTML chargée par un navigateur comme Netscape, elle est appelée une applet ; dans ce cas le code Java correspondant est téléchargé du serveur où il réside vers le poste client au moment de l'appel de la page ; puis ce code est interprété sur le client par l'interpréteur Java, avec des contrôles poussés et des droits réduits afin d'empêcher le code importé de corrompre l'environnement local.

D'autre part, une application Java peut être lancée comme une application standalone (ou autonome) en invoquant directement l'interpréteur Java sur la machine cible. L'application n'est pas alors soumise aux règles de sécurité qui restreignent l'accès des applets aux ressources système.

Pour chaque règle décrite par la suite, on précise si elles sont applicables pour un projet quelconque, ou uniquement en applet ou en standalone.

3. DOMAINE D'APPLICATION

Ce document n'est ni un cours ni un manuel de référence Java. Il s'adresse à des lecteurs qui, sans être experts, ont une bonne connaissance du langage et des bibliothèques de classes associées.

Les règles définies ici visent à faciliter pour chaque projet la mise en œuvre des techniques Java ; elles forment un ensemble de principes de base qui permettent de capitaliser l'expérience acquise sur les projets en évitant d'engager systématiquement une réflexion approfondie sur des sujets consensuels. Cependant de nombreuses règles laissent aux utilisateurs un certain degré de liberté quant à leur application. Chaque projet devra donc compléter les règles de ce manuel de façon à les adapter à la taille et à la nature de l'application, ainsi qu'aux contraintes associées au contexte d'utilisation de la technologie.

Ce document est complété par le document « Règles communes pour l'utilisation des langages de programmation » (DA2).

Pour utiliser les règles JAVA sur un projet, il faudra suivre la procédure suivante :

? Sélectionner dans les règles communes et les règles JAVA, les règles applicables au projet en fonction des critères de tailorisation ; cette sélection s'effectuera avec l'outil de tailorisation.
? Adapter certaines règles au projet.

Le document est ainsi destiné à plusieurs types de lecteurs :

? le chef de projet qui doit spécifier correctement l'application JAVA à développer,
? le chef de projet et/ou l'ingénieur qualité qui doit sélectionner les règles et éventuellement adapter et compléter les règles en fonction du contexte de réalisation,
? les personnes chargées de la réalisation du projet : elles doivent appliquer les règles retenues pour chaque étape du cycle de vie du logiciel.

Le développement d'une applet Java est la plupart du temps considéré comme une partie du développement d'un serveur Web qui doit respecter les règles énoncées dans le document DR3.

4. DOCUMENTS

4.1. DOCUMENTS DE REFERENCE

DR	Identification	Titre
(DR1)	RNC-ECSS-Q-ST-80	Software Product Assurance
(DR2)	RNC-CNES-E-HB-40-509	Règles et recommandations pour l'utilisation du formalisme UML
(DR3)	RNC-CNES-E-HB-40-505	Règles et recommandations pour la réalisation d'un serveur WEB

4.2. DOCUMENTS APPLICABLES

DA	Identification	Titre
(DA1)	RNC-CNES-E-HB-40-508	Démarche de développement objet pour les logiciels
(DA2)	RNC-CNES-Q-HB-80-501	Règles communes pour l'utilisation des langages de programmation.

4.3. AUTRES DOCUMENTS

? Programmation Java (J-F Macary, C. Nicolas - Eyrolles)
? Java in a Nutshell (David Flanagan - O'Reilly & Associates, Inc.)
? Livre d'Or Java (Patrick Longuet - Sybex)
? Thinking in Java (Bruce Eckel - Prentice Hall)
? Practical Java (Peter Haggar - Addison-Wesley)
? http://java.sun.com
? http://www.ibm.com/developer/java
? http://gamelan.earthweb.com

5. TERMINOLOGIE

5.1. GLOSSAIRE

Terme	Définition
API	Application Programming Interface : interface de programmation pour le développement d'une application (attributs et méthodes accessibles).
Applet / Appliquette	Composant applicatif téléchargeable, qui est inclus dans une page HTML et qui s'exécute au sein d'un navigateur Web intégrant une machine virtuelle Java. De tels composants peuvent par exemple être des composants de l'interface utilisateur ou bien des composants pouvant communiquer avec d'autres applications distantes. Les applets sont écrits en Java.
Bean	Les JavaBeans permettent de créer des composants logiciels réutilisables qui peuvent être assemblés en utilisant un outil visuel d'assemblage. Ils sont basés sur une spécification permettant leur découverte automatique (capacité d'introspection) et leur interopérabilité.
final : attribut finalisé	On ne peut pas modifier la valeur de cet attribut. C'est ainsi que l'on définit les constantes en Java.
final : classe finalisée	Une classe final est une classe dont on ne peut pas hériter.
final : méthode finalisée	Une méthode final est une méthode qui ne peut pas être redéfinie dans une sous-classe.
finalize	Nom de la méthode appelée à la libération d'un objet, à l'appel de la méthode System.runFinalization() et éventuellement à la terminaison d'une application Java. Cette méthode peut être redéfinie.
HTML	HyperText Markup Language. Langage de balisage dérivé de SGML, utilisé pour formater les documents publiés sur le Web. HTML repose sur l'utilisation de balises (" tags ") de formatage permettant également l'inclusion de liens hypertextes et d'éléments externes tels que des images et des documents multimédias.
HTTP	HyperText Transfer Protocol. Protocole de communication utilisé pour transporter des documents hypertextes et d'autres documents, entre un serveur et un navigateur Web.
IDE	Integrated Development Environment. Environnement de développement complet (exemple : JBuilder).
IIOP	Internet Inter-ORB Protocol. Protocole de communication développé par l'OMG (Object Management Group) pour l'interopérabilité d'applications au travers du standard CORBA.

Internet	Le réseau Internet est un ensemble mondial de réseaux interconnectés reposant sur le protocole de communication TCP/IP. Ce réseau maillé, relativement tolérant aux pannes, permet le routage dynamique des informations. Le terme Internet est aussi fréquemment utilisé pour désigner les technologies reposant sur TCP/IP, telles que le Web, le courrier électronique, les groupes de discussion,...
Java Plugin	Composant développé par Sun Microsystems, installable dans un navigateur Web et permettant de disposer d'une nouvelle version de machine virtuelle Java. Son utilisation permet de régler les problèmes de compatibilité entre les différentes versions de machines virtuelles fournies avec les navigateurs Web. Le Java Plugin peut être installé dans les navigateurs Netscape et Microsoft à partir de leur version 3.0. Il est disponible pour toutes les versions du JDK à partir de la 1.1.4.
JDK	Java Development Kit. Environnement fourni par Sun Microsystems permettant le développement d'applications Java.
JNI	Java Native Interface. Voir native.
JVM	Java Virtual Machine. Interpréteur de programme Java, compilé dans du byte-code, généralement stocké dans un fichier .class. Lae principe de la JVM permet de concevoir des applications par essence portables.
Machine virtuelle	Voir JVM.
native	Une méthode native est une méthode Java dont l'implémentation est faite dans un autre langage (C, C++,…), au travers d'une API normalisée basée sur C appelée JNI depuis le JDK 1.1.
Navigateur Web	Application utilisée pour localiser et afficher des pages Web et permettant la navigation de page en page en suivant les liens hypertextes.
package	Unité d'organisation des programmes Java, dont la structure de nommage est hiérarchique. Les packages permettent de gérer les visibilités des noms et les unités de compilation.
Ramasse-miettes	Mécanisme chargé de détecter les objets qui ne sont plus utilisés par le programme et de libérer la mémoire correspondante. Le ramasse-miettes (garbage collector) est activé dans un thread de bas niveau de type démon.
Script (JavaScript et VBScript)	Portions exécutables d'un fichier HTML, à la différence des balises qui commandent la présentation des informations. Ces portions utilisent un langage interprété par le navigateur. JavaScript est d'origine Netscape, tandis que VBScript est d'origine Microsoft.
SandBox (Bac à sable)	Architecture permettant d'enfermer le code Java (utilisé en particulier dans les navigateurs Web). La SandBox isole un programme Java des ressources de la machine sur laquelle il s'exécute et des ressources réseau.

Serveur Web	Ordinateur ou application faisant office de serveur et proposant des services Web. Le rôle principal d'un serveur Web est de renvoyer des pages Web en réponse aux requêtes HTTP des navigateurs Web. Un serveur Web est le contenant d'un site Web. Les règles et recommandations pour un serveur Web appartiennent au domaine du génie logiciel.
standalone	Qualifie une application autonome pouvant fonctionner indépendamment, en particulier sans navigateur Web ou appletviewer (contrairement aux applets).
SSL	Protocole au-dessus du protocole IP qui fournit authentification, cryptage et intégrité. (Secured Socket Layer définit par Netscape).
Sérialisation	La sérialisation est un mécanisme offert par Java depuis le JDK 1.1 permettant de transformer un objet (ou plutôt un graphe d'objet) en un flux d'octets représentant son état. Ce mécanisme peut être utilisé pour gérer la persistance des objets, le passage d'objets sur le réseau, …
super	Mot clé permettant d'accéder à l'objet courant en tant que représentant de la classe de niveau supérieur dans l'arbre d'héritage (classe " mère ").
synchronized	Le mot clé synchronized permet de définir des portions de code ou des méthodes comme étant non accessibles simultanément par deux threads sur un même objet ou une même classe (dans le cas de méthodes static synchronized). Ce mécanisme s'appuie sur la manipulation d'un moniteur disponible pour chaque objet ou classe Java. Lorsqu'un thread exécute un bloc synchronized sur un objet, il est impossible à un autre thread d'exécuter une méthode synchronized (la même ou une autre) sur le même objet.
transient	Le mot clé transient permet de définir un attribut dans une classe comme non pris en compte par le mécanisme de sérialisation par défaut de Java.
URL	Uniform Resource Locator. Adresse d'un document, d'un élément inclus ou de toute autre ressource sur le Web. L'URL est formée de plusieurs parties, précisant notamment le protocole utilisé, l'adresse du serveur et la référence de l'élément sur ce serveur (par exemple, http://www.company.com/index.html).
volatile	Le mot clé volatile permet d'identifier les attributs dans une classe qui peuvent être modifiés de manière asynchrone (i.e. par un autre thread pouvant ou non s'exécuter sur un autre processeur) pour lesquels le compilateur ne devra pas effectuer d'optimisation, pour éviter des problèmes d'incohérence lors de l'utilisation de machines multi-processeurs, ou dans le cas d'optimisation faite par le compilateur dans un contexte multi-thread (recopie dans la pile locale d'un thread d'une valeur d'attribut par exemple).
World Wide Web / Web / WWW	Le World Wide Web est un réseau global de services HTTP, FTP, GOPHER, WAIS, faisant partie du réseau Internet. Le World Wide Web est composé principalement de serveurs HTTP d'informations hypertextes accédés par des clients, les navigateurs Web.

5.2. ABREVIATIONS

5.2.1. Codification des règles

Voir DA2

5.2.2. Autres abréviations ou acronymes

Voir DA2

6. ADEQUATION EVENTUELLE AUX NORMES ET STANDARDS

Les règles présentées dans ce document sont conformes à la version 1.3 du JDK de JAVA.

7. LES REGLES

7.1. CONCEPTION / ORGANISATION DU CODE

Org.Importation	Il faut expliciter les classes importées.
M=3;F=0;P=67;V=2	
Quelconque	

Description

Lors de l'importation de classes d'un autre package, il est recommandé de ne pas utiliser le caractère " * " mais plutôt d'indiquer explicitement quelles classes sont utilisées.

Justification

L'importation d'un package entier ne diminue pas l'efficacité du code : en effet seules les classes réellement utilisées sont prises en compte par le compilateur. Cependant une référence exacte aux classes externes utilisées améliore la lisibilité du code.

Exemple

la ligne :

```
import javax.swing.*;
```

ne donne aucune information sur les classes réellement utilisées dans le code. Au contraire les lignes :

```
import javax.swing.JButton;
import javax.swing.JContainer;
```

donnent des informations sur la nature du code développé.

7.2. PRESENTATION DU CODE

Pr.JavaDoc	Chaque programme doit définir la documentation minimale associée à chaque entité.
M=2;F=0;P=62;V=2	
Quelconque	

Description

L'utilitaire javadoc permet de générer une aide en ligne sous forme de pages HTML à partir de commentaires insérés dans le code. Il faut commenter le code en suivant les spécifications de javadoc. Cette règle présente la syntaxe minimale associée à l'usage de javadoc. Cet usage doit être précisé en début de Projet. On s'appuiera notamment sur les règles Pr.CommentDev, Pr.GestionConf et Pr.JavaDocDetail pour compléter la syntaxe obligatoire.
Outre la présentation générale des fonctionnalités de chaque classe, méthode et attribut, les tags suivants doivent impérativement être utilisés quand ils sont pertinents : @author, @see, @param, @return et @exception.
Les commentaires doivent être explicites et on utilisera si besoin le fait que javadoc autorise les commentaires de plusieurs lignes pour les 5 tags cités.

Justification

L'aide en ligne est standardisée ce qui facilite son utilisation ; on peut référencer par des liens hypertexte l'aide en ligne des classes Java.
Cette règle permet de fournir le minimum d'informations nécessaires à la compréhension d'un programme.

Exemple

Exemple 1 : documentation minimale des classes

```
/**
 * Semantique de la classe.
 * @see           Classe ou methode pouvant servir de reference.
 * @see           Autre classe pouvant servir de reference.
 * @author        Auteur de la classe.
 */

// NB: @see ne sera utilise que si necessaire.
```

Exemple 2 : documentation minimale des méthodes

```
/**
 * Semantique de la méthode.
 * @param  1er  parametre de methode.
 * @param  2eme parametre de methode.
 * @return valeur de retour.
 * @exception   exception renvoyée.
 */
```

Exemple 3 : documentation minimale des attributs

```
/**
 * Description de l'attribut.
 * @see    Classe, méthode ou variable pouvant servir de reference.
 */

// NB: @see ne sera utilise que si necessaire.
```

Exemple 4 : documentation du code

```
// On utilise generalement les commentaire style C++ : "//".

// Ceci décrit le contenu du bloc de code qui suit
// ou encore de l'instruction qui figure sur la ligne suivante.

int maVariable; //Ceci commente la déclaration d'une variable locale.
```

Exemple 5 : documentation du code mort

Le code mort n'est acceptable dans un programme que de façon temporaire. On utilisera de préférence les commentaires de style C pour mettre de coté un bloc de code temporairement non utilisé (ces commentaires pourront être retrouvés lors des revues de codes de manière à ne pas les laisser dans le logiciel final).

```
/*
    // Test de debuggage : maVariable ne doit jamais depasser 32 bbp.
    if (maVariable >= 32) {
        System.out.println("Attention, la variable est superieure a 32
!");
    }
*/
```

Exemples complémentaires

```
/**
 * Utilisations du tag @see :
 * @see NomDeClasse
```

```
 * @see com.cnes.image.ConversionImage
 * @see com.cnes.image.ConversionImage#bmp2gif(File, File)
 */

// Utilisation du tag @exception
/**
 * Calcul de la surface d'un rectangle.
 *
 * @param  cote1 longueur du premier cote
 * @param  cote2 longueur du second cote
 * @return la surface rectangle
 * @exception IllegalArgumentException IF (cote1 < 0 || cote2 < 0) Cette
              exception est levee si les dimensions du rectangle ne
              sont pas conformes (longueur < 0).
 */
public float calculerSurfaceRectangle(float cote1, float cote2)
             throws IllegalArgumentException {
    ...
}
```

Pr.CommentDev	Les commentaires javadoc doivent être utilisés pendant les phases de développement.
M=2;F=1;P=84;V=0	
Quelconque	

Description

javadoc propose un ensemble de tags permettant de tracer les évolutions du code lors des phases de développement. Le projet devra préciser les conditions d'utilisation de ces tags.

Justification

En utilisant les tags @bug, @review, @todo et @idea tout au long des phases de développement, on facilite grandement le suivi et la maintenance.

Exemple

```
// Le tag @bug peut etre utilise dans les entetes de classes,
// inferfaces ou methodes.
/**
 * Exemple de methode contenant un bug identifie.
 * @param variable   Description de la variable.
 * @bug              Explication du probleme connu et des cas
d'utilisation
                    dans lesquels il se manifeste.
 */
public methodeBuggee(Variable variable) { ... }

// Les tags @review, @todo et @idea peuvent etre utilises tout au long
// du code pour indiquer un probleme ou une amelioration a effectuer.
// Tous ces tags contiennent un nom d'utilisateur responsable des
// modifications a effectuer.

// Exemples d'utilisation :
```

```
/**
 * @review martinG   La boucle de traitement peut etre optimisee.
 */
...
/**
 * @idea nadineM  Ce probleme pourrait etre resolu de maniere plus
                  elegante en utilisant l'algorithme de Viterbi.
 */
...
/**
 * @todo martinG  Il faut implementer la mise a jour des attributs.
 */
...
```

Pr.GestConf	Les tags javadoc doivent être utilisés pour inclure les notions de gestion de configuration dans la documentation liée au code.
M=2;F=1;P=83;V=2	
Quelconque	

Description

javadoc propose un ensemble de tags permettant de tracer les évolutions liées à la gestion de configuration. Le projet devra préciser les conditions d'utilisation de ces tags.

Justification

En utilisant les tags @version, @deprecated et @since tout au long des phases de développement, on facilite grandement le suivi et la maintenance.

Exemple

```
// Les tags @since et @version sont utilises de preference pour des
// classes ou des interfaces; ils indiquent la version correspondant
// a la creation de la classe et sa version actuelle.
// On pourra utiliser des tags geres automatiquement par les outils
// de gestion de configuration du projet (ex: $Id:$ avec RCS ou CVS)

/**
 * Egalisation d'histogramme sur des images au format BMP.
 *
 * @author   MartinG
 * @see      com.cnes.image.EgalisationTIFF
 * @since    TI-CNES v1.3 (05/03/97)
 * @version  $Id:$
 */
public class EgalisationBMP extends FiltrageFichierImage {
    ...
}
```

Pr.JavaDocDetail	Une bonne connaissance des formalismes javadoc est nécessaire.
M=0;F=0;P=110;V=0	
Quelconque	

Description

Il faut bien connaître les potentialités de javadoc de manière à accroître la réutilisabilité de certains modules. Ainsi, lorsque le code est jugé complexe ou réutilisable, il faut compléter la syntaxe de base de la règle Pr.JavaDoc. Chaque projet définira les conditions précises associées au respect de cette règle.
On pourra notamment définir les pré-conditions, les post-conditions et les problèmes de concurrence. Il est également fortement conseillé d'utiliser certains tags HTML.

Justification

Les tags @precondition ou @pre, @postcondition ou @post et @concurrency permettent de compléter la documentation des classes et de leurs méthodes. Le formatage HTML des commentaires permet d'améliorer la lisibilité de la documentation générée par javadoc.
En respectant cette règle, on facilite donc le suivi et la maintenance.

Exemple

```
/**
 * Application du filtrage gaussien.
 *
 * Exemple d'utilisation :
 * <CODE>
 *      ImageCouleur image = new ImageCouleur("nomFichier.img");
 *      FiltrageGaussien filtre = new FiltrageGaussien();
 *      filtre.setImage(image);
 *      filtre.appliquer();
 * </CODE>
 *
 * @precondition   L'image de reference doit avoir ete definie.
 * @postcondition  Le fichier image est ferme.
 * @concurrency SEQUENTIAL La methode ne peut pas etre utilisee en
parallele.
 *
public void appliquer() {
    ...
}

// Il est possible de formater n'importe quelle description avec
// une syntaxe HTML identique a celle utilisee pour les pages Web.

/**
 * On peut <em>par exemple</em> inserer une liste :
 * <ol>
 * <li> premier point ;
 * <li> deuxieme point ;
 * <li> troisieme point.
 * </ol>
 */
```

7.3. IDENTIFICATEURS

Id.Widget	Les noms des widgets sont des noms de variables ; il faut de plus les préfixer ou les suffixer par leur classe.
M=2;F=0;P=45;V=1	
Quelconque	

Description

Sans Objet

Justification

Sans Objet

Exemple

```
okButton, fileMenu, clientMenuItem    // anglais.
boutonOk, menuFichier, itemMenuClient // français.
```

Id.Package	Le nom des packages doit indiquer leur localisation ou leur appartenance.
M=2;F=1;P=36;V=0	
Quelconque	

Description

Sans Objet

Justification

Ceci permet d'obtenir une classification prenant en compte la localisation et l'appartenance de ces package.

Exemple

```
javax.swing
java.security.interfaces
org.omg.stub.java.rmi
fr.capgemini.www.gestion.fichiers
fr.cnes.qtis.egalisation.spot5
```

7.4. DONNEES

Don.IntInstance	Une classe de base dont on veut interdire l'instanciation doit être définie comme une classe abstraite.
M=1;F=1;P=40;V=1	
Quelconque	

Description

Une classe abstraite en Java est une classe qui ne peut pas être instanciée car elle a été déclarée abstract.

Une classe peut être déclarée abstract sans posséder de méthode abstraite. Par contre, toute classe possédant au moins une méthode abstract devra être déclarée abstract.
De plus, toute classe héritant d'une classe abstraite contenant une ou plusieurs méthodes abstraites devra-t-elle aussi être déclarée abstraite si cette sous-classe ne donne pas une implémentation à chacune des méthodes abstraites de la super-classe.

Justification

Le compilateur empêche toute création d'instance à partir d'une classe abstraite.

Exemple

Supposons que l'on manipule différentes classes représentant des formes : Cercle, Rectangle, Triangle. Chacune de ces classes comporte deux méthodes permettant de trouver l'aire et la circonférence d'une instance.
Pour pouvoir manipuler un tableau de ces formes et obtenir de façon indifférenciée leur aire, on définit une super-classe, Forme, dont héritent Cercle, Rectangle et Triangle, et qui définit la signature de la méthode getAire().
Cependant cette méthode ne peut être définie au niveau de la classe Forme qui ne représente en elle-même aucune forme : cette méthode est une méthode abstraite. Aucune instance de Forme ne peut être créée : cette classe est une classe abstraite.

```
// Classe abstraite.
public abstract class Forme {
    // Methode abstraite.
    public abstract double getAire();
}

// Premiere classe derivee.
public class Cercle extends Forme {
    private double rayon;
    private static final double PI = 3.141598;
    public Cercle() { rayon = 1.0; }
    public Cercle(double r) { rayon = r; }

    // Calcul de l'aire d'un cercle.
    public double getAire() { return PI*rayon*rayon; }
}

// Autre classe derivee.
public class Rectangle extends Forme {
    private double largeur;
    private double hauteur;
    public Rectangle() { largeur = 0.0; hauteur = 0.0; }
    public Rectangle(double l, double h) { largeur = l; hauteur = h; }

    // Calcul de l'aire d'un rectangle.
    public double getAire() { return largeur*hauteur; }
}
```

Don.InterfaceEstUn	La notion d'interface doit implémenter une relation conceptuelle " est un " ou la relation conceptuelle " est une implémentation de ".
M=1;F=0;P=40;V=0	
Quelconque	

Description

Une interface est un regroupement d'opérations définissant un comportement pour toute classe qui l'implémente. Cette notion d'interface se retrouve dans des systèmes comme CORBA ou DCOM. Une classe peut implémenter plusieurs interfaces, et une interface peut hériter de plusieurs autres interfaces. Lorsqu'une classe implémente une interface, elle doit donner une implémentation à toutes les méthodes de cette interface pour être instanciable (condition vérifiée par le compilateur). La notion d'interface correspond donc à une notion de conception par contrat.

Justification

Sans Objet

Exemple

```
// Definition de l'interface d'un vehicule
// Seule la signature des methodes est donnée
// On pourrait prefixer les methodes du mot-cle abstract
// mais ce n'est pas necessaire, les methodes des interfaces
// étant par definition abstraites.

interface Vehicule {
    boolean demarrerMoteur();
    void arreterMoteur();
    float accelerer(float acc);
    boolean tourner(Direction dir);
}

// Definition de la classe Automobile qui definit toutes
// les methodes de l'interface : elle implemente l'interface.

class Automobile implements Vehicule {
    ...
    boolean demarrerMoteur() {
        if (pasTropFroid) {
            moteurDemarre = true;
        } ...
    }

    void arreterMoteur() {
        moteurDemarre = false;
    }

    float accelerer(float acc) {
        ...
    }
    boolean tourner(Direction dir) {
        ...
    }
    ...
```

```
}

// Definition d'une autre classe qui implemente l'interface :
// la classe Camion. Les methodes de l'interface sont definies
// de façon tout a fait independante de leur implementation dans
// Automobile.
class Camion implements Vehicule {
    ...
    boolean demarrerMoteur() { ... }
    void arreterMoteur() { ... }
    float accelerer(float acc) { ... }
    boolean tourner(Direction dir) { ... }
    ...
    // Methodes specifiques.
    void releverBenne() { ... }
}

// Utilisation de ces classes.
Automobile auto = new Automobile();
Camion camion = new Camion();
Vehicule vehicule;

// auto implemente Vehicule et peut etre considere de ce type.
vehicule = auto;
vehicule.demarrerMoteur();
vehicule.arreterMoteur();
...
// camion implemente Vehicule et peut etre considere de ce type.
vehicule = camion;
vehicule.demarrerMoteur();
vehicule.arreterMoteur();
```

Don.Interface	La notion d'interface Java doit être utilisée à bon escient.
M=2;F=0;P=41;V=0	
Quelconque	

Description

Il ne faut pas abuser des interfaces.

Justification

Les interfaces permettent d'établir des relations transversales dans les arbres d'héritage de l'application. Ces " râteaux " sont pratiques et souvent nécessaires, par exemple pour mettre en œuvre des " callbacks " ou pour accéder à des données applicatives dans le code d'une IHM. Ils portent cependant atteinte à la pureté de la conception objet de l'application.
Une classe peut implémenter plusieurs interfaces. Le code peut rapidement devenir difficile à comprendre et à maintenir si on abuse de cette possibilité.

Exemple

L'utilisation d'une interface est particulièrement intéressante pour définir un " service " commun à plusieurs objets de conception différente.
Considérons par exemple deux classes : une classe Affichage faisant partie d'une IHM et permettant d'afficher du texte, et une classe Source générant le texte à afficher. Source est un fournisseur de texte ; cette classe pourrait stocker une référence à un objet de type Affichage, mais cela empêcherait Source d'envoyer du texte à un objet d'un autre type (par exemple à une Imprimante) ; il faudrait alors avoir une sous-classe de Source pour chaque type de destinataire pour pouvoir les gérer.
Une solution plus élégante consiste à définir une interface utilisable par Affichage et par toute autre classe manipulant du texte, et à stocker une référence à cette interface dans Source. L'interface se nomme TexteModifiable et est implémentée dans la classe Affichage (et dans d'autres classes comme Imprimante). Un objet de type Affichage peut alors être utilisé partout où l'on a besoin d'un TexteModifiable.

```
// Definition de l'interface.
interface TexteModifiable {
    recevoirTexte(String texte);
}

// Implementation de l'interface.
class Affichage implements TexteModifiable {
    ...
    public recevoirTexte(String texte) {
        scrollText(texte);
    }
    ...
}

// Utilisation de l'interface.
class Source {
    TexteModifiable destinataire;
    Source(TexteModifiable dest) {
        destinataire = dest;
    }
```

```
    private envoyerTexte(String texte) {
        destinataire.recevoirTexte(texte);
    }
    ...
}
```

Du point de vue fonctionnel, la seule chose qui importe pour la classe Source est d'avoir accès à une méthode pour envoyer du texte. L'objectif de l'interface est ici de garantir que tout type d'objet l'implémentant fournira cette méthode (recevoirTexte).

Don.MéthAbstr	Il faut préférer les méthodes abstraites aux méthodes de corps vide.
M=2;F=0;P=42;V=1	
Quelconque	

Description

Une méthode pour laquelle on ne peut pas définir d'implémentation valide doit être déclarée abstraite plutôt que d'être définie avec un corps vide (renvoyant une valeur par défaut).
A noter que certaines méthodes peuvent avoir de façon légitime un corps vide lorsque leur sémantique est " ne rien faire " (par exemple, une rotation d'un cercle autour de son centre).

Justification

Il vaut mieux définir la méthode comme abstraite car cela impose de réfléchir à nouveau à sa mise en œuvre lors de la définition d'une classe dérivée.

Exemple

Exemple incorrect

Dans l'exemple de la règle Don.IntInstance, si on ne définit pas de classe abstraite Forme mais une classe de base avec une méthode vide on a :

```
// Classe de base.
public class Forme {
    // Methode vide.
    public double getAire() { return 0.0; }
}

// Forme derivee.
public class Cercle extends Forme {
    ...
    // Surcharge de la methode getAire.
    public double getAire() { return PI*rayon*rayon; }
}

// Autre Forme derivee.
public class Rectangle extends Forme {
    ...
    // La methode getAire n'est pas redefinie
    // celle de la classe Forme s'applique !
}
```

On peut instancier Forme (contraire à la règle Don.IntInstance) et décrire une classe dérivée qui oublie de définir une méthode getAire() correcte.

Exemple correct

Dans certains cas, il peut être nécessaire de définir des méthodes vides.
Dans l'exemple suivant, la classe Cercle qui hérite de la classe Figure doit fournir une implémentation de la méthode rotation() (puisque cette méthode est déclarée dans Figure). Cette méthode possède légitimement un corps vide.

```
// Classe de base.
public class Figure {
    ...
    public abstract void rotation(double angle);
    ...
}

// Classe dérivée.
public class Cercle extends Figure {
    ...
    // Methode vide.
    public void rotation(double angle) {}
}
```

Don.MembreStatic	On ne doit pas abuser des classes ou interfaces membres statiques.
M=3;F=0;P=31;V=1	
Quelconque	

Description

Java permet de définir une classe à l'intérieur d'une autre classe et de lui associer le mot clé static ; il s'agit d'une classe interne statique (static inner class). De la même façon on peut définir une interface à l'intérieur d'une classe (static inner interface); le mot clé static est alors implicite.
Les inner classes peuvent être définies comme static lorsqu'elles sont déclarées dans le corps de la classe imbriquante – c'est à dire ni dans une méthode ni imbriquée, comme le JDK le permet, dans une autre inner class –. Leur portée peut être précisée à l'aide des mots clés public, private, protected.
Les classes ou interfaces internes statiques déclarées private peuvent être utilisées pour rendre plus lisible l'implémentation d'une classe complexe.
Les classes internes statiques non private (i.e. public, " visibilité package " ou protected) doivent être utilisées avec plus de réserve. On pourra notamment les utiliser pour coder les énumérations en Java.

Justification

L'utilisation de classes internes statiques déclarées private (comme celle des classes internes non statiques) permet de structurer le code d'implémentation d'une classe complexe, ce qui le rend plus lisible (voir Don.Inner).
L'intérêt des classes ou interfaces internes statiques déclarées non private est simplement de pouvoir regrouper des classes proches sémantiquement.
Bien que la notion de package réponde déjà à ce besoin, il est parfois nécessaire d'avoir des rapprochements entre classes à un niveau plus bas que celui des packages. C'est notamment le cas pour la définition de constantes énumérées (voir exemple).
Le cas des constantes énumérées mis à part, il faut veiller à limiter le plus possible l'utilisation de classes imbriquées non privées. En effet, le code source résultant est moins lisible car apparaît au niveau de la structure des classes une imbrication qui ne correspond à aucune imbrication des services ni à aucune coopération particulière entre la classe incluse et la classe imbriquante.

Exemple

Exemple incorrect

```
// La classe ComportementSocial est une classe incluse
// definie comme statique.

class Animal  {
    static public class ComportementSocial {
        ...
    }
    ...
}
```

La classe ComportementSocial se comporte exactement comme une classe de niveau supérieur et de nom Animal.ComportementSocial. On peut en créer une instance de la façon suivante :

```
Animal.ComportementSocial meute = new Animal.ComportementSocial();
```

Il vaut mieux définir la classe ComportementSocial de façon indépendante.

Exemple correct

```
// La classe Frequence est une classe incluse definie comme statique
// qui permet de definir une constante enumeree typee.

class Journal  {
    static public class Frequence {
        // Permet de limiter l'utilisation du constructeur.
        private Frequence() {}
    }
    ...
    public final static Frequence MENSUEL     = new Frequence();
    public final static Frequence TRIMESTRIEL = new Frequence();

    public Journal(Frequence frequence) {
       ...
    }
}
```

La classe Frequence permet de définir une constante énumérée que l'on peut utiliser de la façon suivante :

```
Journal j = new Journal(Journal.MENSUEL);
```

Don.Inner	On ne doit pas abuser des classes incluses non statiques.
M=2;F=0;P=32;V=1	
Quelconque	

Description

Java permet de définir des classes à l'intérieur d'une autre classe, et même à l'intérieur de n'importe quel bloc de code. Ces classes sont appelées inner classes (classes incluses ou internes).
Dans le cas où la classe incluse est définie au niveau des attributs de la classe imbriquante, on dit qu'il s'agit d'une classe membre (member class). Dans le cas où la classe incluse est définie dans un bloc de code on dit qu'il s'agit d'une classe locale (local class).
Dans le cas où une classe membre n'est pas définie comme statique (voir règle Don.MembreStatic), l'imbrication de la classe membre dans la classe imbriquante signifie :

l'accès aux membres de la classe imbriquante par la classe membre ;
qu'à une instance d'une classe imbriquante correspond une instance de la classe membre.
On peut même définir les classes membre et les classes locales de façon anonyme (anonymous inner classes).
Il faut utiliser cette facilité à bon escient.

Justification

Les classes incluses permettent dans certains cas de rendre le code bien plus synthétique, et plus lisible, en permettant d'éviter la définition d'une classe externe si elle ne doit être utilisée qu'une seule fois ; les classes internes anonymes sont très utiles dans la gestion des événements graphiques du JDK.
Cependant si la classe incluse est de taille importante on obtient l'effet inverse (code illisible) ; le fait que la classe hôte et la classe imbriquée peuvent réciproquement utiliser les services qu'elles définissent, suggère plutôt une coopération entre classes du même package.

Exemple

Exemple 1 (member class)

```
// La classe Cerveau est incluse dans la classe Animal. On fait l'hypothese
// que seuls les animaux possedent un Cerveau et que les services rendus par
// une instance de la classe Cerveau ne peuvent pas etre utilises en dehors
// du contexte de l'instance d'Animal qui possede ce Cerveau.

// Tous les objets de type Cerveau sont crees dans une instance de la classe
// Animal, le constructeur de Cerveau n'etant accessible que depuis Animal.

Class Animal {
    Class Cerveau {
        ...
    }
    // Definition d'une methode accessible depuis Animal et Cerveau.
    void protegerTerritoire() { ... }
}
```

Exemple 2 (local class)

```
// La classe Cerveau est incluse dans la methode protegerTerritoire
// Il faut bien sur que le Cerveau ne serve qu'a proteger son
// Territoire, sinon il va falloir redefinir le Cerveau ailleurs.

Class Animal {
    void protegerTerritoire() {
        Class Cerveau {
            ...
        }
    }
}
```

Exemple 3 (anonymous inner class)

L'exemple suivant montre la définition d'une classe interne anonyme à l'intérieur de la méthode getEnumeration(). La classe interne est définie en même temps que l'instance dans l'instruction return. Cette classe interne implémente l'interface Enumeration et à ce titre définit les méthodes abstraites hasMoreElement et nextElement.
L'instance renvoyée est la seule instance existante de cette nouvelle classe non nommée ; le type de retour déclaré pour la méthode getEnumeration() est le type Enumeration qu'implémente la classe non nommée – on serait bien en peine de déclarer que la méthode renvoie une instance de la nouvelle

classe car elle n'a pas de nom, mais cela n'empêche pas d'accéder aux services de cette nouvelle classe qui sont l'implémentation des méthodes abstraites de l'interface.

```
Enumeration getEnumeration() {

    return new Enumeration() {
        int element = 0;
        boolean hasMoreElements() {
            return element < employees.length;
        }
        Object nextElement() {
            if (hasMoreElements()) {
                return employees[element++];
            } else {
                throw new NoSuchElementException();
            }
        }
    };
}
```

On remarque que ce code est peu lisible, ce qui est imputable à la taille de la classe incluse.

Exemple 4 (anonymous inner class)

L'exemple suivant est plus judicieux et montre l'utilisation des classes incluses dans la gestion des événements graphiques ; on note la concision du code.

```
// La classe anonyme est une classe derivee de la classe MouseAdapter
// et surcharge la methode mouseClicked en assignant a l'evenement
// clic souris l'action pertinente – dans ce cas handleClicks.

addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) { handleClicks(e); }
});
```

Don.MembreVis	Il est interdit de modifier la visibilité des classes membres.
M=3;F=1;P=25;V=1	
Quelconque	

Description

La visibilité par défaut d'une classe membre est celle du paquetage.
Java permet d'associer un indicateur de visibilité à la définition d'une classe membre (public, protected, private), ce qui suppose une modification de la visibilité par défaut de cette classe.
Il ne faut pas modifier la visibilité par défaut des classes membre.

Justification

Donner une portée public à une classe incluse n'est pas naturel (pourquoi l'inclure ?).
De plus les classes incluses déclarées comme protected sont en fait traitées par le JDK comme des classes de portée public ce qui rend particulièrement pernicieux l'utilisation de protected dans ce cas.
Enfin les classes incluses déclarées comme private sont en fait traitées par le JDK comme des classes de la portée du paquetage : il faut là encore éviter cette ambiguïté.

Exemple

Sans Objet.

Don.TableauxDecl	Il faut déclarer les tableaux de façon uniforme.
M=2;F=1;P=28;V=2	
Quelconque	

Description

Il faut déclarer les tableaux en associant les crochets au type de la façon suivante :
int[] tableau;
plutôt que :
int tableau[];
Il ne faut surtout pas mélanger les notations.
Remarque : la notation Java (int[] tableau) améliore la lisibilité du code.

Justification

Le code est plus homogène donc plus lisible.
Il est aussi plus satisfaisant de regrouper les indicateurs de type avant la variable : on note ainsi int[] le type tableau d'entiers.

Exemple

Exemple incorrect

```
// rangee et colonne sont des tableaux d'octets.
// matrice est un tableau de tableau d'octets.
byte[] rangee, colonne, matrice[];

// Cette methode prend en parametre un tableau d'octets et un
// tableau de tableaux d'octets.
public void exemple(byte[] colonne, byte[] matrice[]) { ... }
```

Exemple correct

```
byte[] rangee;
byte[] colonne;
byte[][] matrice;
public void exemple(byte[] colonne, byte[][] matrice) { ... }
```

Don.AllocNull	Il est inutile de tester l'allocation des objets.
M=0;F=0;P=81;V=1	
Quelconque	

Description

En Java, il n'est pas nécessaire de tester que le retour d'une création d'objet est un pointeur NULL.

Justification

Cette habitude provenant des développeurs C++ n'est pas nécessaire en Java et surcharge inutilement les applications.

Si une allocation n'est pas possible, c'est la machine virtuelle qui se charge de lever une exception. Au besoin, cette exception peut être traitée localement ou globalement par l'application.

Exemple

Sans Objet.

Don.List	On ne doit pas utiliser la classe Vector et choisir le type de classe "liste" le plus adapté en fonction de l'algorithme.
M=2;F=0;P=78;V=1	
Quelconque	

Description

Quatre types d'implémentation de liste sont présents dans le JDK : ArrayList, LinkedList, Vector et le tableau d'éléments (array). De manière à optimiser les performances, il est souhaitable d'effectuer les choix suivants :
Si la taille du tableau est fixe, on utilisera un tableau simple (array). Le tableau possède en outre l'avantage de contenir des données typées et d'éviter ainsi les transtypages intempestifs.
Pour une liste classique, il faut utiliser par défaut la classe ArrayList. Cette dernière fournit en effet les meilleures performances d'accès aux données.
Dans le cas particulier où l'algorithme nécessite un grand nombre d'insertions et de suppressions, la classe LinkedList pourra être employée.
On évitera donc l'utilisation de la classe Vector qui fournit les mêmes services que la classe Arraylist mais avec des performances amoindries (sa présence dans le JDK est historique et son utilisation ne doit être envisagée que pour des raisons de compatibilité antérieure à Java 2).

Remarque : La classe Vector est synchronisée alors que ArrayList ne l'est pas. Cela signifie que l'on ne peut pas utiliser directement un objet de type ArrayList s'il doit être accédé par des threads concurrents. Dans ce cas il faut par exemple encapsuler la liste via la méthode Collections.synchronizedList(list).

Justification

Obtenir une implémentation adaptée aux performances souhaitées et garantir la pérennité de l'application développée.

Exemple

Exemple de performances obtenues avec les différentes implémentations d'une liste :

Type	Get	Iteration	Insert	Remove
array	1430	3850	---	---
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

Le tableau ci-dessus représente le temps nécessaire (en ms) à l'exécution de 50 000 opérations. Ces performances ne sont données qu'à titre indicatif et peuvent bien évidemment varier en fonction de la plate-forme et de la machine virtuelle utilisées.

Don.Ensemble	Il faut adapter le type de classe " ensemble " implémenté en fonction de l'application visée.
M=1;F=0;P=100;V=0	
Quelconque	

Description

Deux implémentations d'ensembles sont présentes dans le JDK : TreeSet et HashSet. Les ensembles gérés par la classe HashSet ont en général des performances meilleures que ceux gérés par la classe TreeSet (particulièrement pour les ajouts et les recherches). Il est donc préférable d'utiliser HashSet. La seule raison pouvant entraîner le choix de la classe TreeSet est la nécessité d'avoir constamment des données triées.

Justification

Obtenir une implémentation adaptée aux performances souhaitées.

Exemple

Exemple de performances obtenues avec les différentes implémentations d'un ensemble :

Type	Taille	Ajout	Recherche	Itération
TreeSet	10	138.00	115.00	187.0
	100	189.50	151.10	206.50
	1000	150.60	177.40	40.04
HashSet	10	55.00	82.00	192.00
	100	45.60	90.00	202.20
	1000	36.14	106.50	39.39

Le tableau ci-dessus représente le temps nécessaire (en ms) à l'exécution de 50 000 opérations sur 10 éléments en fonction en de la taille des ensembles. Ces performances ne sont données qu'à titre indicatif et peuvent bien évidemment varier en fonction de la plate-forme et de la machine virtuelle utilisées.

Don.Dictionnaire	Il faut éviter d'utiliser la classe `Hashtable` et choisir le type de classe " données indexées " en fonction de l'application visée.
M=1;F=0;P=99;V=0	
Quelconque	

Description

Le JDK fournit 3 implémentations d'ensembles de données indexées : Hashtable, HashMap et TreeMap.
De manière à optimiser les performances, il est souhaitable d'effectuer les choix suivants :
utiliser par défaut la classe HashMap ;
n'utiliser TreeMap que dans le cas exceptionnel où les données doivent être constamment triées ;
limiter l'utilisation de la classe Hashtable à la résolution de problèmes de compatibilité antérieure à Java 2.

Les performances des classes Hashtable et HashMap sont quasiment équivalentes en général. On préférera cependant HashMap car ses performances sont légèrement supérieures et son architecture, introduite avec Java 2, est amenée à perdurer.

Justification

Obtenir une implémentation adaptée aux performances souhaitées et garantir la pérennité de l'application développée.

Exemple

Exemple de performances obtenues avec les différentes implémentations d'un ensemble de données indexées :

Type	Taille	Déposer	Prendre	Itération
TreeMap	10	143.00	110.00	186.00
	100	201.10	188.40	280.10
	1000	222.80	205.20	40.70
HashMap	10	66.00	83.00	197.00
	100	80.70	135.70	278.50
	1000	48.20	105.70	41.40
Hashtable	10	61.00	93.00	302.00
	100	90.60	143.30	329.00
	1000	54.10	110.95	47.30

Le tableau ci-dessus représente le temps nécessaire (en ms) à l'exécution de 50 000 opérations sur 10 éléments en fonction en de la taille des ensembles. Ces performances ne sont données qu'à titre indicatif et peuvent bien évidemment varier en fonction de la plate-forme et de la machine virtuelle utilisées.

Don.BitSet	L'utilisation de la classe `BitSet` est interdite.
M=1;F=0;P=101;V=1	
Quelconque	

Description

La classe BitSet permet de stocker des valeurs binaires sous forme compacte mais, au regard de ses performances, son usage est à déconseiller.

Justification

Les performances de manipulation d'un tel ensemble sont bien inférieures à celles obtenues avec un tableau classique. Le seul intérêt de cette classe est l'éventuel gain mémoire mais il faut savoir qu'un objet de ce type aura une taille minimum de 64 bits. L'intérêt de l'usage de cette classe est donc fortement discutable.

Exemple

Sans Objet.

7.5. TRAITEMENTS

Tr.ConstrBut	Il faut réserver les constructeurs aux tâches d'initialisation.
M=1;F=0;P=36;V=0	
Quelconque	

Description

Sans Objet

Justification

Assigner aux constructeurs d'autres tâches que celles d'initialisation nuirait à la compréhension du programme et peut amener des pertes de performance : le constructeur est par définition la méthode appelée à la création de l'objet.

Exemple

Sans Objet.

Tr.ConstRetour	Il ne faut pas définir de " constructeur " avec une valeur de retour.
M=1;F=1;P=35;V=2	
Quelconque	

Description

Java permet de définir une méthode de même nom que la classe et avec une valeur de retour ; mais attention : ce n'est alors pas un constructeur !

Justification

C'est à éviter car cela peut conduire à des erreurs de programmation pendant le développement et à une mauvaise compréhension du code pendant la maintenance.

Exemple

Exemple Incorrect

```
public class ClasseSansConstructeur {
    // Attributs.
    private static int compteur;

    // Methode de nom ClasseSansConstructeur.
    public int ClasseSansConstructeur(int increment){
        compteur = compteur + increment;
        return compteur;
    }

    // Autres methodes.
    ...
}
```

Tr.DestrFinalize	Il ne faut pas redéfinir la méthode `finalize()`.
M=1;F=1;P=31;V=2	
Quelconque	

Description

Le langage Java ne permet pas de définir de destructeur. C'est le système de ramasse-miettes qui permet de libérer la mémoire allouée pour les objets qui ne sont plus utilisés. Cependant certains objets utilisent d'autres ressources, qui ne sont pas traitées par le ramasse-miettes : descripteurs de fichiers, identifiants de sockets, etc.
On peut coder la libération de ces ressources dans la méthode finalize(), qui sera exécutée soit implicitement à la libération de l'objet, juste avant que le traitement ramasse-miettes ne soit effectué, soit par l'appel explicite à la méthode System.runFinalization().
Cependant la redéfinition de la méthode finalize() pour libérer des ressources est fortement déconseillée. Il vaut mieux gérer explicitement la libération des ressources en utilisant par exemple le bloc finally de la série try/catch/finally, pour être sûr que même en cas d'exception les ressources sont libérées.

Justification

Les libérations de ressources seront effectuées lors du prochain ramassage de miettes ; un seul ennui : on ne sait pas lorsque celui-ci aura lieu, on n'est même pas sûr qu'il y en aura un avant que l'interpréteur ne se termine. Cette incertitude rend le code non déterministe !

Exemple

Prenons le cas où la ressource libérée dans la méthode finalize() est un descripteur de fichier. La méthode finalize() est appelée. Un peu plus loin dans le code on essaie de supprimer le répertoire contenant le fichier. C'est l'échec car le ramasse-miettes n'a pas été appelé entre-temps, le fichier n'est donc pas fermé, le répertoire ne peut pas être supprimé.

Tr.SuperFinalize	En cas de redéfinition de la méthode `finalize()` on doit appeler `super.finalize()`.
M=1;F=2;P=28;V=2	
Quelconque	

Description

Il n'y a pas de chaînage automatique des appels à la méthode finalize(), comme c'est le cas pour les constructeurs par défaut.
Si on définit la méthode finalize(), il faut appeler explicitement celle de la super-classe à la fin de cette méthode.

Justification

Les traitements de terminaison de la classe de plus haut niveau, s'ils existent, doivent être appelés.

Exemple

```
// Destructeur de la classe FileOutputStream.
// Ferme le stream quand le ramasse-miettes est invoque.
```

```
// Verifie qu'il n'est pas deja ferme.
protected void finalize() throws IOException {
    if (fd != null) {
        close();
    }
}

// Classe MaClasseFichier.
public class MaClasseFichier extends FileOutputStream {

    protected void finalize() throws IOException {
        // Traitements specifiques a MaClasseFichier.
        ...
        // Appel au destructeur de la super classe.
        super.finalize();
    }
}
```

Tr.RazReferences	Il faut remettre à `null` les références aux objets volumineux comme les tableaux.
M=0;F=2;P=45;V=1	
Quelconque	

Description

null représente en Java une référence non établie.
Lorsqu'un objet de grande taille (par exemple, un tableau) n'est plus utilisé par une application, on peut permettre sa destruction en déréférençant l'objet. Ceci est réalisé en forçant toutes les références sur l'objet à null ou à tout autre objet).

Justification

Cela permet de libérer plus vite (au prochain lancement du ramasse miettes) des ressources mémoire qui ne servent plus.

Exemple

```
// Declaration d'un tableau de grande taille.
int[] tableau = new int[100000];

// Utilisation de ce tableau.
int resultat = calculsDivers(tableau);

// On n'a plus besoin du tableau. Apres l'instruction suivante
// comme le tableau n'est plus reference nulle part la
// prochaine activation du ramasse miette liberera la memoire allouee.
tableau = null;
```

Tr.AffectMask	Aucune affectation ne doit être masquée par une autre opération.
M=3;F=1;P=12;V=1	
Quelconque	

Description

Il ne faut pas effectuer une affectation en même temps qu'un appel de fonction et au sein d'une condition ou d'une expression.

Justification

Le cumul d'opération sur une même ligne d'instruction nuit fortement à la lisibilité des programmes et risque de masquer des opérations.

Exemple

Exemple incorrect

```
...
if (resultat = calculValeur(increment++)) {
    nbResultatsNonNuls++;
}
...
```

Exemple correct

```
...
resultat = calculValeur(increment);
increment++;
if (resultat != 0) {
    nbResultatsNonNuls++;
}
...
```

Tr.MéthodePublic	Une méthode ne doit être `public` que si elle doit être utilisée depuis au moins une classe définie dans un autre package.
M=2;F=0;P=18;V=1	
Quelconque	

Description

La portée d'une méthode doit par défaut être la portée private. Dans une classe, on ne définira public une méthode que si elle est utilisée dans une autre classe non dérivée.
Attention cependant à ne pas aller à l'encontre de la réutilisation. Une méthode ne doit pas être déclarée private si elle correspond à un service fourni par la classe (même si ce service n'est pas utilisé dans la version courante du projet).

Justification

Dans un souci de clarté, d'efficacité et de facilité de maintenance il faut limiter au maximum l'interface de programmation de chaque classe.

Exemple

Sans Objet.

Tr.MéthodeProtected	Une méthode ne doit être `protected` que si elle doit être utilisée depuis au moins une méthode définie dans une autre classe de la sous-hiérarchie.
M=2;F=0;P=19;V=1	
Quelconque	

Description

La portée d'une méthode doit par défaut être la portée private. Une méthode ne doit être protected que si elle est utilisée par une classe qui dérive de celle où elle est définie.

Justification

Dans un souci de clarté, d'efficacité et de facilité de maintenance il faut limiter au maximum l'interface de programmation de chaque classe.

Exemple

Sans Objet.

Tr.MéthodeRedef	Les définitions d'une fonction membre redéfinie entre une classe de base et une classe dérivée doivent comporter les mêmes noms de paramètres.
M=2;F=2;P=22;V=1	
Quelconque	

Description

Sans Objet

Justification

Cela assure une meilleure lisibilité des programmes.

Exemple

```
public class Voiture {

    // Methode permettant de dessiner une voiture.
    public void dessiner(String couleur) {
        // Dessiner la voiture
    }
}
public class Formule1 extends Voiture {
    // Methode dessiner redefinie pour la classe Formule1.
    // Le parametre definissant la couleur porte le même nom
    // que dans la classe de base.
    public void dessiner(String couleur) {
        // Appel a la methode dessiner de la classe Voiture.
        super.dessiner(couleur);
    }
}
```

Tr.MéthodeFinal	Il faut préfixer du mot clé `final` les méthodes dont on veut interdire la redéfinition.
M=2;F=1;P=25;V=1	
Quelconque	

Description

On peut vouloir interdire la redéfinition d'une méthode sur de nombreux critères :
elle est très utilisée,
elle a un fort niveau de complexité fonctionnelle et/ou algorithmique,
elle utilise des ressources critiques,
ses performances sont critiques,
elle n'a pas, sémantiquement, de raison d'être redéfinie.
Associer le mot clé final à cette méthode en interdit la redéfinition.

Justification

La méthode redéfinie risque d'être de moindre utilité ou de moindre qualité que la méthode initiale ; il faut absolument éviter que les utilisateurs de classes dérivées n'utilisent la méthode redéfinie en pensant utiliser la méthode de la classe de base.
Il vaut donc mieux créer une nouvelle méthode.

Exemple

Sans Objet.

Tr.MéthodeSurcharge	Les différentes surcharges d'une fonction doivent offrir des services ayant la même sémantique.
M=3;F=1;P=17;V=0	
Quelconque	

Description

Sans Objet

Justification

Il s'agit d'être cohérent avec la notion de service associé à une méthode.

Exemple

Exemple incorrect

```
public class Voiture {
    // Definir la puissance de la voiture.
    public void definir(Cheval chevaux) {
        ...
    }
    // Definir la couleur de la voiture.
    public void definir(Couleur couleur) {
        ...
    }
}
```

Les deux méthodes de l'exemple ci-dessus ne devraient pas porter le même nom car elle n'ont apparemment pas la même sémantique et sûrement pas les mêmes répercussions sur l'objet.

Exemple correct

```
public class Voiture {
    // Dessin a partir du nom de la couleur.
    public void dessiner(String couleur) {
        ...
    }
    // Dessin a partir du triplet RGB.
    public void dessiner(int rouge, int vert, int bleu) {
         ...
    }
}
```

Tr.This	Il faut utiliser la référence `this` à l'objet courant avec parcimonie.
M=1;F=0;P=67;V=2	
Quelconque	

Description

this doit être utilisé :
lorsque l'on souhaite passer une référence à l'objet courant à un autre objet,
lors de l'appel d'un constructeur depuis un autre constructeur de la même classe,
pour lever une ambiguïté de portée de variable ou de méthode.
Il ne doit pas être utilisé pour l'appel de méthodes de l'objet courant.

Justification

Pour appeler une méthode de la classe courante la syntaxe habituelle est de ne pas préfixer le nom de la méthode par this afin de ne pas alourdir le code.

Exemple

```
class Enfant {
    private String nomEcole;
    private String prenom;
    private String nom;

    public Enfant(String prenom, String nom, String ecole) {
        // Appel de l'autre constructeur.
        this(prenom, nom);
        nomEcole = ecole;
    }

    public Enfant(String prenom, String nom) {
        // Le mot cle this permet de differencier le parametre de la
        // methode et l'attribut sans pour autant avoir a utiliser
        // des noms de parametres moins explicites.
            this.prenom = prenom;
            this.nom = nom;
     }

     public void passerObjet() {
          // Le mot cle this est utilise pour passer une reference
```

```
                // a l'objet courant.
                fournirAcces(this);
            }
        }
```

Tr.SwitchBreak	Tout branchement d'un choix multiple doit se terminer par l'instruction `break`.
M=3;F=1;P=15;V=2	
Quelconque	

Description

Une instruction permettant de sortir de la structure switch doit être utilisée à la fin de chaque branchement (on utilisera généralement break et parfois return ou continue).

Justification

Un branchement à choix multiple ne doit normalement conduire qu'à l'exécution d'un seul bloc d'instruction.
L'absence de break conduit en Java à l'exécution de toutes les instructions suivant l'entrée validée en pratique, cette absence résulte donc très majoritairement d'une erreur de codage.

Exemple

```
switch (choix) {
    case 1:
        traitement_1();
        break;
    case 2:
        traitement_2();
        traitement_3();
        break;
    default:
        throw new Exception("traitement d'un choix non prévu");
}
```

Tr.Blocs	Tous les blocs d'une instruction de contrôle doivent être délimités par des accolades.
M=3;F=1;P=19;V=2	
Quelconque	

Description

Pour délimiter les lignes de code liées aux instructions if, for, while et do, on utilisera systématiquement les accolades ouvrantes et fermantes. Cette règle doit être appliquée même si le bloc ne contient qu'une seule instruction.

Justification

Cette règle simplifie la maintenance du code en facilitant l'ajout d'opérations dans les instructions de contrôle, elle assure une meilleure lisibilité en uniformisant les syntaxes. Notons également que le fait d'ouvrir systématiquement une accolade après une instruction de contrôle prévient de l'apparition d'un point virgule parasite (bug particulièrement fréquent et délicat à détecter).

Exemple

Exemple incorrect

```
for (noImage = 0; noImage < nombreImages; noImage++)
    for (noColonne = 0; noColonne < nombreColonnes; noColonne++)
      for (noLigne = 0; noLigne < nombreLignes; noLigne++)
            tableauImages[noImage].pixels[noColonne][noLigne] = noImage;
```

Exemple correct

```
for (noImage = 0; noImage < nombreImages; noImage++) {
    for (noColonne = 0; noColonne < nombreColonnes; noColonne++) {
        for (noLigne = 0; noLigne < nombreLignes; noLigne++) {
            tableauImages[noImage].pixels[noColonne][noLigne] = noImage;
        }
    }
}

// Une instruction peut etre ajoutee sans risque.

for (noImage = 0; noImage < nombreImages; noImage++) {
    int valeurPixel = noImage + 1;
    for (noColonne = 0; noColonne < nombreColonnes; noColonne++) {
        for (noLigne = 0; noLigne < nombreLignes; noLigne++) {
            tableauImages[noImage].pixels[noColonne][noLigne] =
valeurPixel;
        }
    }
}
```

Tr.InstanceOf	Il faut utiliser l'opérateur `instanceOf` avec parcimonie.
M=3;F=0;P=14;V=2	
Quelconque	

Description

L'opérateur instanceOf peut être appliqué à tout objet.
Il permet de tester si une instance est bien une instance de la classe qui est le deuxième opérande de instanceOf.
Il peut être utilisé pour effectuer des traitements différents selon la classe ou la sous-classe d'un objet.
Cependant, ce type d'aiguillage qui rappelle les instructions switch() sur un champ définissant le type en langage C doit être évité au maximum.

Justification

Cela peut nécessiter des modifications de code si l'arbre d'héritage est enrichi; supposons qu'on ait au départ une classe de base M et des classes dérivées F1, F2, F3 ; un objet de classe effective F1, F2 ou F3 peut être référencé comme un objet de classe M et dans un cas particulier testé grâce à l'opérateur instanceOf pour introduire des traitements spécifiques. Si on ajoute une classe F4 dérivée de la classe M, il faut ajouter du code pour traiter le cas F4 même si aucun traitement nouveau n'est à définir ; ce code est externe à la nouvelle classe ; ceci augmente de façon inutile le couplage entre les classes.
Ceci peut être utilisé à la place du mécanisme de surcharge pour effectuer dans une classe de base des traitements différents suivant la sous-classe effective d'un objet donné. Dans ce cas cela relève d'une mauvaise compréhension des mécanismes d'héritage et de surcharge des méthodes : on ne tire pas parti de la puissance du langage orienté objet.

Exemple

Sans Objet.

Tr.Iterations	Il faut réaliser les itérations avec l'interface `Iterator`.
M=2;F=1;P=49;V=1	
Quelconque	

Description

Pour parcourir de manière itérative les différents ensembles d'objets, le JDK contient deux interfaces spécifiques : Iterator et Enumeration. Outre le fait de fournir des fonctionnalités adaptées au parcours itératif, ces interfaces permettent de s'affranchir du type d'implémentation utilisée pour l'ensemble des données à traiter ; leur utilisation est donc fortement conseillée.
On utilisera par contre systématiquement l'interface Iterator au détriment de l'interface Enumeration.
Beaucoup de collections d'objets peuvent être parcourues à la fois par un Iterator et une Enumeration, il faut toujours préférer l'interface Iterator. Certains objets n'offrent qu'un parcours par l'interface Enumeration, par extension ces derniers sont donc à proscrire (une implémentation plus récente de leur(s) fonctionnalité(s) est généralement présente dans le JDK (utiliser par exemple, HashMap plutôt que Hashtable et ArrayList plutôt que Vector).

Justification

L'interface Iterator introduite par Java 2 est préférable à l'ancienne interface Enumeration car elle est plus rapide et offre des fonctionnalités accrues. Le respect de cette règle garantit la pérennité de l'application développée et optimise l'implémentation.

Exemple

```
// Methode de suppression des elements null.
public void supprimeObjetsNull(Iterator iteration) {
    while (iteration.hasNext()) {
```

```
            if (iteration.next() == null) {
                iteration.remove();
            }
        }
    }

    // Exemple d'appel de cette methode.
    ArrayList maliste;
    ...
    supprimeObjetsNull(maListe.iterator());
    ...
```

7.6. GESTION DES ERREURS

Err.SousClasException	Les exceptions du projet doivent être des sous-classes de la classe `Exception`.
M=2;F=0;P=26;V=1	
Quelconque	

Description

Les trois classes de base prédéfinies pour les exceptions sont :
java.lang.Error : cette classe sert à décrire les problèmes d'édition de liens dynamique, les problèmes de la machine virtuelle comme le manque de mémoire, etc.
java.lang.RuntimeException : cette classe décrit des erreurs d'exécution susceptibles d'être générées par toute portion de code, comme ArrayOutOfBoundException.
java.lang.Exception : cette classe sert de base à toutes les autres descriptions d'erreur, en particulier pour les erreurs applicatives.
Les classes qui servent à décrire les exceptions du projet doivent être dérivées de la classe java.lang.Exception.
De plus, une documentation exhaustive des exceptions levées est indispensable.

Justification

Il est plus lisible de réserver les classes java.lang.Error et java.lang.RuntimeException aux erreurs " système ". Cela peut permettre de traiter de façon spécifique ces erreurs.

Exemple

```
    // Definition d'une exception utilisee pour les
    // erreurs de saisie.
    public class SaisieException extends Exception {
        public void afficher(Frame f) {
            FenetreErreur fenetre = new FenetreErreur(f);
            fenetre.afficher(getMessage());
        }
    }

    // Exemple de generation de l'exception
    // par la methode saisieValeur.
    public int saisieValeur() throws SaisieException {
        int saisie = 0;   // Valeur saisie par l'operateur.
```

```
        ...
        if (saisie > 100) {
           throw new SaisieException("Saisir une valeur entre 1 et 100");
        }
        return saisie;
    }

    // Exemple de traitement de l'exception.
    ...
    try {
        int valeur = saisieValeur();
    } catch (SaisieException e) {
         e.afficher(frame_courant);
    }
```

Err.ExcepSpecific	Il ne faut pas manipuler directement les classes `Exception` et `RunTimeException`.
M=1;F=1;P=26;V=1	
Quelconque	

Description

On ne doit jamais utiliser les instructions catch et throws avec les types de base Exception et RunTimeException. Ces types peuvent par contre être dérivés et utilisés dans le respect de la règle Err.SousClasException.

Justification

Le fait de capter une exception de type Exception est beaucoup trop général et masque irrémédiablement la spécificité de toutes les classes dérivées. Il ne faut donc traiter que les classes dérivées et ne jamais présumer de l'existence d'exceptions étrangères au module algorithmique courant.

Exemple

Exemple incorrect

```
    // Utilisation de l'exception et de la methode de la regle
    // "EXCEP-SousClasse".

    try {
        int valeur = saisieValeur();
    } catch (Exception e) {
        if (e instanceOf SaisieException) {
            (SaisieException)e.afficher(frameCourant);
        } else {
            System.err.println("Exception inconnue !");
            System.exit(1);
        }
    }
```

Exemple correct

```
    // Utilisation de l'exception et de la methode de la regle
    // "EXCEP-SousClasse".

    try {
        int valeur = saisieValeur();
```

```
} catch (SaisieException e) {
    e.afficher(frameCourant);
}
```

Err.ClassError	Il ne faut pas capter les exceptions de la classe `Error`.
M=0;F=1;P=50;V=2	
Quelconque	

Description

Il faut laisser ces exceptions se propager jusqu'au plus haut niveau ; l'interpréteur Java imprime alors un message d'erreur sur la sortie standard et se termine.

Justification

Ces exceptions sont levées suite à des erreurs en général considérées comme irrattrapables.

Exemple

InternalError : erreur interne d'un (mauvais) interpréteur Java.
NoSuchMethodError : appel d'une méthode qui n'existe pas dans la version chargée de la classe.

Err.RuntimeExcep	Il ne faut pas explicitement propager les exceptions de la classe `RuntimeException`.
M=1;F=3;P=29;V=1	
Quelconque	

Description

Il faut parfois traiter les RuntimeExceptions explicitement.
Les exceptions de cette classe n'ont pas besoin d'être déclarées avec la méthode qui les génère pour se propager, éventuellement jusqu'au plus haut niveau, ce qui peut amener la machine virtuelle à les traiter elle-même. Le traitement par défaut de la machine virtuelle d'une exception de ce type est d'interrompre le thread qui l'a généré (par exemple, dans le cas d'une application mono-thread, l'application sera interrompue !). Les portions de code réellement susceptibles de renvoyer ce type d'exception devront être gérées par des blocs try/catch/finally.

Justification

Ces exceptions sont susceptibles d'être générées par n'importe quelle portion de code Java. Les traiter toutes explicitement alourdirait énormément le code.
Par contre, pour certaines parties sensibles du code (qui manipulent les tableaux, les cast, les entrées-sorties,…, qui mettent en cause des interactions de l'utilisateur, ou qui font appel à des portions de code en cours d'écriture), il est judicieux de traiter ces exceptions.

Exemple

Exemple incorrect

```
System.out.println("  Entrer une valeur numérique ");
String s = br.readLine();
int i = Integer.parseInt(s);
```

Exemple correct

```
System.out.println("  Entrer une valeur entière ");
try {
    String s = br.readLine();
    int i = Integer.parseInt(s);
}
```

```
catch(NumberFormatException e) {
    System.out.println(" Votre valeur n'est pas entière, désolé ");
    System.exit(0);
}
```

7.7. DYNAMIQUE

Dyn.ThreadCreation	Il faut utiliser l'interface `java.lang.Runnable` pour créer un thread.
M=1;F=0;P=80;V=2	
Quelconque	

Description

On préférera créer une implémentation de l'interface java.lang.Runnable plutôt que de créer une sous-classe de java.lang.Thread.

Justification

Dans l'acception purement objet de la dérivation de classes, créer une sous-classe de java.lang.Thread signifierait créer un nouveau type de Thread, ce qui n'est pas du tout ce que l'on fait lorsqu'on dit qu'une classe permet de créer un thread.
De plus la méthode de création de threads utilisant les interfaces permet une meilleure conception orientée objet ; en effet Java ne permet pas l'héritage multiple mais permet l'implémentation d'interfaces multiple, donc implémenter l'interface Runnable, ce qui est du domaine de la mise en œuvre technique, n'a pas de conséquence sur le reste du design.
La méthode run() a de toutes façons toutes les chances d'être sémantiquement très liée à l'une des classes de l'application ; choisir la mise en œuvre par implémentation d'interface permet à la méthode run() d'accéder à des membres privés ou protégés de cette classe.

Exemple

Exemple incorrect

```
class Dessin extends Object {
    public void rafraichir() {
    ...
    }
}

class Animation extends Thread {
// Animation ne peut pas dériver de Dessin
    ...
    // Redéfinition de la méthode run().
    public void run() {
        while (true) {
            // Dessin.
            ...
            repaint();
        }
    }
}
```

Dyn.Daemon	Les threads effectuant des traitements de fond doivent être définis comme des “ démons ”.
M=1;F=2;P=45;V=1	
Quelconque	

Description

On appelle ici traitement de fond un traitement qui rend des services techniques de base qui ne correspondent pas à des spécifications fonctionnelles applicatives (l’interpréteur Java en fournit un exemple avec le ramasse-miettes).
On indique qu’un thread est un “ démon ” en utilisant la méthode setDaemon(). Un thread “ démon ” est tué par Java lorsque les autres threads de l’application se terminent.

Justification

On n’a pas à se préoccuper de gérer la fin de ces threads ce qui simplifie le code.
Les threads sont surtout utilisables pour les applications Java standalone, ainsi que pour le système Java lui-même (exemple du ramasse-miettes), mais pas pour les applets. Comme toute applet est créée à l’intérieur d’une autre application Java, les threads “ démons ” qu’elle est susceptible d’activer continueraient à vivre tant que l’application qui contrôle l’applet existe, ce qui n’est probablement pas l’effet désiré.

Exemple

```
// Exemple ou la classe Demon derive de la classe Thread, ce qui
// enfreint la regle THREAD-Creation, mais simplifie l’exemple.
// Le demon est un thread dedie a la gestion des connexions de
// nouveaux clients sur un socket de connexion.

class Demon extends Thread {
    Demon() {
        setDaemon(true);   // Le thread est un "démon".
        start();
    }
    public void run() {
        while (true) {
            // Attente de connexion sur un socket dedie.
            // Si connexion Alors
                // Creation d’un socket pour gerer le nouveau client
                // Activation d’un autre thread
                // pour gerer les entrees-sorties sur ce socket.
            // Finsi
        } // La tâche est désactivée par Java.
    }
}
```

Dyn.Exit	Il faut fermer les threads non “ démons ” avant d’appeler `System.exit()`.
M=1;F=2;P=44;V=1	
Quelconque	

Description

La méthode System.exit() doit être appelée uniquement après avoir stoppé ou terminé les threads importants de l'application.

Justification

Cette méthode force l'interruption de tous les threads en cours de l'application, sans terminer ceux-ci de manière normale. Dans un cas d'erreur, l'utilisateur n'aurait donc aucune chance de rétablir la situation ou plus simplement de ne pas perdre son travail. Il faut donc toujours traiter le cas d'erreur en cours avant d'appeler cette méthode.

Exemple

On peut utiliser System.exit() si le cas d'erreur est irrécupérable et si on a tenté de la décrire clairement dans un message informant l'utilisateur.
Cette méthode peut également être utilisée pour de petits programmes se comportant comme des commandes du système d'exploitation destinés à être utilisé dans des scripts de commandes.

Dyn.TempsRéel	Il ne faut pas utiliser Java pour les applications à forte contrainte temps réel.
M=1;F=1;P=10;V=2	
Quelconque	

Description

Sans Objet.

Justification

Même s'il est possible de désactiver le ramasse miettes qui risque de rendre l'exécution du code non déterministe au niveau de son temps de réponse, il est préférable d'utiliser des environnements Java spécifiques (comme par exemple implémentant la Real-Time Specification for Java) qui sortent du cadre de ce document.

Exemple

Sans Objet.

7.8. INTERFACES

Int.AccesRessource	Il faut utiliser les méthodes `getProperty` et `setProperty`.
M=2;F=1;P=56;V=1	
Quelconque	

Description

Lorsqu'on a besoin d'accéder à des caractéristiques extérieures, ou de paramétrer le comportement de l'application en fonction du contexte d'exécution, il faut utiliser la méthode System.getProperty(...) pour extraire des informations, le passage des informations au moment de l'invocation de l'interpréteur pour en positionner de façon statique, et la méthode System.setProperty(...) pour en positionner de façon dynamique.
Il ne faut pas par exemple définir un fichier de paramétrage de l'application et lire dedans le chemin d'accès au répertoire des données.

Justification

Le mécanisme des propriétés existe déjà. Il faut donc l'utiliser.

Exemple

Passage d'une variable d'environnement comme propriété à l'interpréteur Java (sous Windows) :

```
C:\> java -Dprinter=%PRINTER%
```

Idem (sous UNIX) :

```
% java -Dprinter=$PRINTER
```

7.9. QUALITE

QA.TestUnitaire	Il faut introduire dans les classes développées les méthodes `main`() et `toString`().
M=3;F=2;P=28;V=2	
Quelconque	

Description

main() contiendra le code des tests unitaires et toString() fournira un affichage pertinent de l'état d'un objet de cette classe.
De plus, la méthode main() fournira à terme un exemple opérationnel d'utilisation de la classe.

Justification

Pour faciliter le test et le debuggage des classes.
Notons qu'il est d'usage en Java de pouvoir, à l'aide de la méthode toString(), afficher sous forme de chaîne de caractère l'état (informations pertinentes issues des valeurs des membres) d'un objet (même si ce n'est pas la finalité de ce dernier).
Le code de main() doit rester raisonnable ; il contiendra les tests unitaires surtout pour les classes de base de l'application et non des tests de niveau intégration pour les autres classes.

Exemple

```
public class Cercle extends Forme {
    private double rayon;
    private static final double PI = 3.141598;

    public Cercle() { rayon = 1.0; }
    public Cercle(double r) { rayon = r; }

    // Calcul de l'aire d'un cercle.
    public double getAire() { return PI*rayon*rayon; }

    // Affichage de l'etat d'un objet cercle.
    public String toString() {
        return "Cercle de rayon : " + rayon;
    }

    // Exemple d'utilisation utilise pour la validation de la classe.
    public static void main(String[] args) {
        Cercle cercleRayon1 = new cercle();
        Cercle cercleRayonPI = new cercle(PI);

        System.out.println(cercleRayon1);
        System.out.println("Aire = " + cercleRayon1.getAire());
        System.out.println(cercleRayonPI);
        System.out.println("Aire = " + cercleRayonPI.getAire());
    }
}
```

QA.VersionNavig	Il faut s'assurer que la version des navigateurs utilisés supporte le JDK utilisé.
M=3;F=1;P=20;V=1	
Applet	

Description

Une applet Java est interprétée par l'interpréteur Java intégré dans le navigateur. Cet interpréteur doit être compatible avec le JDK utilisé (Java Development Kit).
Le navigateur peut-être compatible de façon native ou le devenir par l'ajout d'un Plugin permettant d'utiliser une version spécifique du JDK.

Justification

Une applet ne peut fonctionner que si le navigateur est compatible avec la version du JDK utilisée !
L'utilisation du Java Plugin permet de garantir la version de machine virtuelle utilisée.

Exemple

Netscape Navigator 3.0 et Internet Explorer 3.0 ne supportent pas le JDK 1.1.4.
Toutes les versions de Netscape Navigator 4.05 ne supportent pas le JDK 1.1.4.
Netscape Navigator 4.5 supporte le JDK 1.1.5.

QA.PortNative	On ne doit utiliser des méthodes natives qu'en cas de force majeure.
M=3;F=0;P=40;V=1	
Quelconque	

Description

On ne doit jamais utiliser de méthode native (code externe écrit en C ou autre) dans un programme, sauf s'il n'y aucun autre moyen technique possible de réaliser la fonctionnalité voulue. Autre dérogation possible : la réutilisation justifiée de composants logiciels non Java (eg. d'un code serveur d'une application client/serveur) ou un problème crucial de performance (ponctuel).
On devra utiliser des librairies de classes Java fournissant cette fonctionnalité, ou on réécrira celle-ci avec les librairies standard de l'API Java.

Justification

L'utilisation de méthode native implique par nature l'abandon d'un grand nombre des bénéfices de l'utilisation du langage Java : sécurité, indépendance vis à vis de la plate-forme matérielle, ramasse-miettes (garbage collection), chargement automatique de classes via le réseau,...
De plus, un programme qui mixe du code Java et des méthodes natives introduit des trous dans le système de sécurité Java. Il n'y en effet aucune garantie que le code ainsi écrit n'introduise pas de virus, et si un problème de référence mémoire existe dans ce code, comme l'accès à une zone protégée de mémoire, cela provoquera certainement une erreur irrécupérable dans l'interpréteur Java, et stoppera donc le programme.

Exemple

Si l'on ne dispose pas des fonctions voulues, on peut les réécrire en Java.

QA.Officiel	On ne doit utiliser que la partie officielle des APIs utilisées.
M=3;F=1;P=30;V=1	
Quelconque	

Description

Lorsque l'on utilise une API Java de base ou spécialisée (comme Java 2D), on ne doit se servir que des classes et interfaces documentées, c'est à dire celles qui font partie de la spécification officielle.
La spécification officielle est celle dont la documentation javadoc est fournie avec les paquetages ; certains services fournis par les classes des paquetages de base ou spécialisés sont accessibles depuis l'extérieur mais non documentés ; il ne faut pas les utiliser.

Justification

Toute implémentation d'une API Java contient certainement des particularités qui ne sont pas décrites dans les spécifications officielles, voire utilise des spécificités de la plate-forme cible. Les programmes Java ne doivent donc en aucun cas se baser sur ces API pour des raisons de portabilité.
Tout ce qui n'est pas documenté officiellement n'est pas fiable en ce qui concerne la portabilité. S'y fier peut introduire des différences de comportements entre plate-forme qui sont dangereuses et souvent difficiles à détecter.

Exemple

Ne pas définir de sous-classe à l'intérieur d'un package d'une API Java, mais toujours à l'extérieur, sinon celle-ci a de fortes chances d'être dépendante de l'implémentation.
Ne jamais utiliser directement les classes peer du package java.awt.peer car elles dépendent de la plate-forme matérielle GUI de la machine cible.

QA.SystemCall	Il ne faut pas appeler de méthode ouverte au système.
M=3;F=0;P=41;V=1	
Quelconque	

Description

La classe java.lang.Runtime fournit des méthodes permettant d'accéder à la plate-forme matérielle de la machine. On ne doit se servir de ces méthodes qu'en respectant des règles précises de portabilité (à définir dans chaque cas).
Cette règle est applicable pour toutes les méthodes donnant au programme la possibilité d'accéder à des particularités spécifiques à la machine cible.
Cette règle est applicable aux applets car leur environnement d'exécution peut contenir un SecurityManager plus permissif que celui installé par défaut avec les navigateurs classiques.

Justification

Bien que ces méthodes soient portables, elles offrent la possibilité au programmeur peu scrupuleux d'utiliser des caractéristiques de la machine qui ne sont pas portables, rendant du même coup l'application non portable. Il faut donc garantir une norme d'utilisation de ces méthodes, qui sont par ailleurs peu nombreuses. La plus connue est la méthode exec() de la classe Runtime.

Exemple

Dans le cas de la méthode exec() de la classe java.lang.Runtime, on ne doit pas l'utiliser exclusivement par commodité. On pourra par exemple autoriser son usage si et seulement si :

l'invocation est activée le plus directement possible par l'utilisateur et celui-ci doit en être averti.
l'utilisateur a la possibilité de préciser avec l'application les commandes (ou programmes) qui vont être exécutés.
tout dysfonctionnement de cette méthode est géré de la manière la plus rigoureuse possible, par exemple avec l'utilisation d'un traitement d'exception adapté.

QA.PortGetProperty	Il faut utiliser les propriétés portables de la plate-forme si besoin.
M=3;F=1;P=35;V=1	
Quelconque	

Description

La classe java.lang.System fournit les méthodes getProperty et getProperties pour accéder aux constantes dépendantes du système. On les utilisera le plus souvent possible, à moins que des constantes existant dans les classes utilisées n'en fournissent déjà la valeur.
Le JDK 1.3 fournit les propriétés suivantes :

java.version	Numéro de version de Java
java.vendor	Nom de " l'éditeur "

java.vendor.url	URL de “ l’éditeur ”
java.home	Répertoire d’installation de Java
java.vm.specification.version	Version de la spécification de la machine virtuelle
java.vm.specification.vendor	“ Editeur ” de la spécification de la machine virtuelle
java.vm.specification.name	Nom de la spécification de la machine virtuelle
java.vm.version	Version de la machine virtuelle (implémentation)
java.vm.vendor	“ Editeur ” de la machine virtuelle
java.vm.name	Nom de la machine virtuelle
java.specification.version	Version de l’environnement d’exécution Java
java.specification.vendor	“ Editeur ” de l’environnement d’exécution Java
java.specification.name	Nom de l’environnement d’exécution Java
java.class.version	Numéro de version du format des classes Java
java.class.path	Chemin d’accès aux classes de base de Java
Os.name	Nom du système d’exploitation
Os.arch	Architecture du système d’exploitation
Os.version	Version du système d’exploitation
file.separator	Séparateur de fichier ("/" sous UNIX)
path.separator	Séparateur du Path (":" sous UNIX)
line.separator	Séparateur des lignes ("\n" sous UNIX)
user.name	Nom du compte de l’utilisateur courant
user.home	Répertoire “ home ” de l’utilisateur courant
user.dir	Répertoire de travail courant

Remarque : pour des raisons de sécurité les applets ne peuvent pas lire les propriétés suivantes : java.home, java.class.path, user.name, user.home, et user.dir.

Justification

Ces valeurs ont été définies par les concepteurs du langage Java, afin de permettre au programmeur de paramétrer dynamiquement le code suivant la plate-forme matérielle, le système d’exploitation, et la machine virtuelle Java utilisée. Utiliser ces méthodes pour obtenir les propriétés dépendantes de la machine garantit avec certitude la valeur de ces constantes, ainsi que leur portabilité.

Exemple

```
System.getProperty("line.separator");
```

QA.AppProperty	Il ne faut jamais redéfinir des propriétés de l’application dépendantes de la plate-forme.
M=3;F=1;P=37;V=1	
Quelconque	

Description

Java permet aux applications de définir leur propres propriétés (qui s’ajoutent à celles définies par la classe java.lang.System). On ne définira jamais de propriété de l’application qui dépendrait de la plate-forme matérielle, sauf si cette propriété n’est pas définie par la plate-forme Java de base.

Justification

Le mécanisme de propriété (property) défini par Java permet aux applications de définir un fichier d’initialisation ou de paramétrage utilisé par celles-ci pour adapter dynamiquement leur fonctionnement.

Utiliser cette possibilité pour introduire une dépendance vis à vis de la plate-forme ne doit pas recouvrir des définitions de propriétés déjà fournies par l'environnement de base Java, sinon il en résulterait des incohérences potentielles entre ces deux types de propriétés.
Les propriétés de l'application ne doivent donc concerner que sa partie spécifique et ne jamais occulter, même indirectement, les propriétés fournies en standard.

Exemple

Sans Objet.

QA.GUISize	On ne doit jamais fixer en dur la taille et/ou la position des widgets.
M=3;F=1;P=46;V=1	
Quelconque	

Description

Lorsque l'on crée une interface graphique en Java, il ne faut jamais assigner de manière statique la taille ou la position des composants graphiques.

Justification

Bâtir la présentation d'une interface sur des valeurs fixes ou assignées de manière impérative par le programme n'est pas portable.
On devra plutôt mixer de façon judicieuse des classes de LayoutManager adaptées au type de fenêtre à afficher, en s'arrangeant pour que les tailles et positions des composants soient calculées dynamiquement par ces LayoutManagers lors de l'instanciation des fenêtres.
Le positionnement absolu d'un composant graphique n'est accessible que par une utilisation détournée des classes du JDK. Il est cependant parfois utilisé par des environnements de développement. Malgré leur apparente complexité, les politiques de placement du JDK sont prévues pour faciliter l'adaptation des IHM aux différentes résolutions des stations hôtes.

Exemple

Exemple incorrect

```
public void init() {
    myFrame = new JFrame();
    myFrame.resize(100, 100);
    JButton myButton = new JButton(?Ok?);
    myButton.reshape(25, 50, 0, 0);
    ...
}
```

Exemple correct

```
public void init() {
    myFrame = new JFrame();
    myFrame.resize(100, 100);
    myFrame.setlayout(new BorderLayout());
    JButton myButton = new JButton(?Ok?);
    myFrame.add(?West?, myButton);
    ...
}
```

QA.GUIFonts	Il ne faut pas coder en dur les polices de caractères utilisées.
M=3;F=0;P=50;V=1	
Quelconque	

Description

Sans Objet

Justification

Sans Objet

Exemple

Sans Objet.

QA.GUIPaintGC	Il ne faut pas enregistrer de `Graphics` à l'intérieur d'une instance.
M=2;F=2;P=20;V=1	
Quelconque	

Description

Bien souvent, les applications ou applets utilisant des fonctions purement graphiques (éditeur d'image ou de dessin, animation dans une applet) ont besoin de l'instance d'objet java.awt.Graphics associé à un composant ; cette instance est une version native de l'objet graphique sous Windows, Motif,... Or les instances de java.awt.Graphics ont une durée de vie gérée par des mécanismes dont le programmeur n'a pas la visibilité. En particulier l'instance de Graphics créée pour une instance de composant peut être supprimée de la mémoire si le composant n'est plus affiché, puis recréée si besoin (par exemple lorsqu'on fait défiler une liste de composants).
Il ne faut pas mémoriser dans une variable de classe ou d'instance un tel objet de type Graphics. Par contre certaines méthodes comme la méthode paint() des composants graphiques, fréquemment redéfinie dans les classes applicatives, garantissent la pérennité d'un Graphics pendant la durée de leur exécution.

Justification

Les instances de ces objets ont une durée de vie très variable à l'intérieur d'un programme, non seulement suivant la plate-forme graphique, mais également suivant le déroulement du programme. Il y a donc beaucoup de risques qu'un tel objet n'existe pas ou soit incomplètement défini, ou simplement invalide lorsqu'on voudra le réutiliser.
Il vaudra mieux s'attacher à récupérer dans une variable locale à une méthode cette instance de Graphics et l'utiliser exclusivement quand on en a besoin. On sait d'autre part que lorsqu'un composant graphique a été construit et est visible, l'instance de Graphics associée est disponible. On aura donc plutôt intérêt à conserver à disposition une référence sur ce composant et lui demander quel est son Graphics courant.
De façon plus précise, la disponibilité des objets Graphics, qui sont vus dans les programmes comme des instances de la classe abstraite Graphics, mais qui sont en fait des instances de classes concrètes dépendantes de la plate-forme graphique, est limitée à la fois dans le temps et suivant les composants de la façon suivante:
La plate-forme graphique qui pour des raisons d'économie en ressource matérielle graphique ou simplement technique en limite la disponibilité à tout instant – Windows95 n'autorise au maximum l'existence que de quatre instances de Graphics –.

Seuls les composants Canvas et Container disposent à coup sûr d'un Graphics associé (également Button dans les futures implémentations du JDK), Il n'existe aucune garantie de la disponibilité d'un tel objet pour les autres composants de l'AWT ; chaque fournisseur d'implémentation de la machine virtuelle est libre de son choix de réalisation des couches basses de l'AWT.
Seules les méthodes Component.paint() et Component.update() permettent de disposer d'un Graphics valide associé si le composant est une instance de Canvas ou Container. C'est donc uniquement dans ces méthodes que l'on disposera d'un Graphics associé au composant qui soit valide.
Une cascade d'appels à l'intérieur des méthodes paint() ou update() peut amener à une saturation des ressources graphiques (en terme de validité des Graphics) de la plate-forme, en particulier si des threads indépendants sont démarrés dans ces méthodes.
L'instance de Graphics associée à un composant graphique est disponible seulement si celui-ci est visible. Il y a donc des risques qu'une référence à un Graphics soit invalide si on la mémorise dans une variable d'instance.
On s'attachera donc à appeler explicitement la méthode Component.getGraphics() sur les composants du type Canvas ou Container qui sont visibles, ou à mémoriser la référence au Graphics passée en paramètre de la méthode paint() ou update() seulement à l'intérieur de celle-ci. Ceci assure la validité de l'instance de Graphics utilisée.

Exemple

Redéfinition de la méthode paint() dans un composant graphique dérivé :

```
Graphics myGraphics = null;
void paint(Graphics g) {
    if (myGraphics == null) {
        myGraphics = g.create;
    }
    // Suit le code pour la methode paint.
}
```

QA.GUIEvent	On ne doit jamais utiliser le modèle événementiel du JDK 1.0.*.
M=3;F=1;P=34;V=1	
Quelconque	

Description

On utilisera toujours le modèle événementiel introduit depuis la version 1.1 du JDK et jamais celui du JDK 1.0. Les méthodes handleEvent, action, mouseDown,... sont proscrites. Il faudra instancier des EventListener adaptés à chaque événement.

Justification

Le modèle événementiel du JDK 1.0 est condamné à disparaître et présente de nombreux inconvénients qu'on ne présentera pas ici.

Exemple

Exemple incorrect

```
public boolean handleEvent(Event event) {
    // On peut sortir de l'application.
    if (event.id == Event.WINDOW_DESTROY) {
        System.exit(0);
```

```
        }
        return super.handleEvent(event);
    }
```

Exemple correct

```
    public class Fenetre extends JFrame implements WindowListener {
        Fenetre() {
            ...
            this.addWindowListener(this);
            ...
        }

        public void windowClosed(WindowEvent event) {
            System.exit(0);
        }
    }
```

QA.Deprecated	Il ne faut pas utiliser les méthodes marquées comme `Deprecated` dans le JDK.
M=3;F=0;P=32;V=1	
Quelconque	

Description

Parmi les évolutions du JDK, on doit noter qu'un certain nombre d'entre elles sont signalées comme deprecated. On n'utilisera pas ces méthodes dans les programmes Java.

Justification

Toutes ces méthodes sont vouées à la disparition dans les versions futures du JDK. Elles présentaient des problèmes de portabilité, d'internationalisation ou de performance. Elles ont été remplacées par des méthodes plus puissantes ou mieux conçues, soit dans les mêmes classes, soit dans des classes spécialisées.
Notons que tant qu'une méthode appartient à une API Java officielle et qu'elle n'est pas marquée comme deprecated, elle sera obligatoirement maintenue dans la prochaine version du JDK. A partir du moment ou une méthode est notée deprecated, elle peut à tout moment disparaître du JDK.

Exemple

Exemple incorrect

```
    Date maDate = new Date(97, 09, 25);
```

Exemple correct

```
    SimpleDateFormat df = new SimpleDateFormat("yyyy,MM,dd");
    try {
        Date maDate = df.parse(value);
    } catch (java.text.ParseException e) {
        ...
    }
```

Il faut noter qu'il existe toujours une alternative pour suppléer aux méthodes deprecated mentionnée dans la documentation du JDK.

QA.Swing	Les classes `swing` doivent toujours être utilisées en priorité par rapport aux classes `awt`.
M=2;F=0;P=80;V=1	
Quelconque	

Description

Dans les premières versions du JDK, un ensemble de classes graphiques a été mis au point en utilisant les composants présents sur toutes les plates-formes cibles. Ces classes, regroupées dans les sous-packages du package java.awt, se sont avérées rapidement insuffisantes. Elles ne garantissaient en outre pas complètement la portabilité des interfaces créées.
Depuis Java 2, de nouvelles classes graphiques ont été ajoutées au JDK. Ces classes sont regroupées dans les nombreux sous-packages du package javax.swing. Les classes swing garantissent la création d'interfaces 100% pur Java. Elles sont bien plus riches que les classes awt et assurent une bien meilleure stabilité de l'aspect des IHM multi-plates-formes.
Il est donc nécessaire d'utiliser systématiquement les classes swing pour créer des interfaces graphiques.

Justification

L'utilisation des classes swing garantit la pérennité de l'application développée et facilite grandement le portabilité de ces dernières.
Rappelons que tous les composants awt possèdent un équivalent swing et que les classes swing sont facilement différentiables par leur préfixe " J " (Frame ✍ JFrame, Applet ✍ Japplet, etc.).

Exemple

Exemple incorrect

```
// Panel de demonstration contenant 2 boutons.
import java.awt.*;

public class Panel2Boutons extends Panel {
    Button premierBouton = new Button("Bouton No1");
    Button secondBouton  = new Button("Bouton No2");

    public Panel2Boutons() {
        setLayout(new FlowLayout());
        add(premierBouton);
        add(secondBouton);
    }
}
```

Exemple correct

```
// Panel de demonstration contenant 2 boutons.

// Import des classes swing et du composant awt necessaires.
import javax.swing.*;
import java.awt.FlowLayout;

public class Panel2Boutons extends JPanel {
    JButton premierBouton = new JButton("Bouton No1");
    JButton secondBouton  = new JButton("Bouton No2");

    public Panel2Bouton() {
        setLayout(new FlowLayout());
        add(premierBouton);
```

```
            add(secondBouton);
        }
    }
```

QA.IHMStable1	Une classe graphique doit être “ stable ”.
M=2;F=3;P=26;V=0	
Quelconque	

Description

La stabilité d’un composant graphique est garantie par le fait qu’il ne lève pas d’exception lors de sa création et qu’il ne nécessite pas de nettoyage lors de sa destruction (pas besoin de redéfinir la méthode finalize()).
De manière à assurer la stabilité d’un composant graphique, il convient en outre qu’il hérite d’un composant stable et qu’il ne soit composé que de composants stables. Notons que toutes les classes swing présentes dans le JDK peuvent être qualifiées de composants stables.

Justification

La création (composition) et la destruction des interfaces utilisateurs sont des phases qui ne sont pas censées nécessiter d’opérations particulières. Un utilisateur de classe graphique doit par exemple pouvoir assembler des composants avant d’en avoir défini le contenu. Le problème des constructeurs stables est particulièrement visible lorsqu’un module IHM visualisant des données, nécessite une initialisation des ces dernières (le composant graphique doit pouvoir être créé avant les données).
Le respect de cette règle facilite notamment la création de JavaBeans.

Exemple

Exemple incorrect

```
public class VisuInfoScene3D extends JTextArea {
    private Scene3D scene;
    // Attention, l’objet Scene3D doit etre initialise avant la
    // construction du visualisateur pour permettre l’execution
    // de la methode informationGenerales().
    public VisuInfoScene3D(Scene3D scene) {
        this.scene = scene;
        setText(scene.informationGenerales());
    }
    ...
}
```

Exemple correct

```
public class VisuInfoScene3D extends JTextArea {
    private Scene3D scene;
    private static final String manqueInfo = "-- Scene Non Initialisée --
";

    public VisuInfoScene3D(Scene3D scene) {
        this.scene = scene;
        if ((scene != null) && (scene.isInit() == true)) {
            setText(scene.informationGenerales());
        } else {
            setText(manqueInfo);
```

```
        }
    }
    ...
}
```

QA.IHMStable2	Si le constructeur d'une classe graphique peut être amené à échouer, une exception doit être levée.
M=2;F=2;P=23;V=1	
Quelconque	

Description

Bien que cela représente une transgression de la règle QA.IHMStable1, si la création d'un composant graphique peut être amenée à échouer, cet échec doit être notifié par une exception.

Justification

Cette règle doit être respectée pour empêcher l'utilisateur du composant de continuer à construire son IHM en pensant que l'initialisation est correcte.

Exemple

Si l'exemple incorrect de la règle QA.IHMStable1 ne peut être rendu correct en testant l'état de l'objet de type Scene3D passé au constructeur (et que l'on ne veut ou ne peut pas modifier la classe Scene3D), le constructeur de la classe VisuInfoScene3D peut être amené à échouer.

```
// NoDataException est une Exception definie dans la librairie 3D
utilisee.
// EchecInitException est une Exception "maison".

public class VisuInfoScene3D extends JTextArea {
    private Scene3D scene;
    // Attention, l'objet Scene3D doit etre initialise avant la
    // construction du visualisateur pour permettre l'execution
    // de la methode informationGenerales(). Le cas echeant, une
    // exception sera levee.
    public VisuInfoScene3D(Scene3D scene) throws EchecInitException {
        try {
            setText(scene.informationGenerales());
            this.scene = scene;
        } catch (NoDataException ndException) {
            throw new EchecInitException();
        }
    }
    ...
}
```

QA.SecurApplet	On doit sécuriser les applets Java.
M=1;F=3;P=10;V=0	
Applet	

Description

Dans tout développement d'applets Java, on doit définir les contraintes de sécurité. On doit vérifier que le comportement sécuritaire associé par défaut au navigateur utilisé est compatible avec ces contraintes de sécurité en ne perdant pas de vue leur impact sur les fonctionnalités de l'application ainsi que sur l'efficacité du développement.
Au besoin il faut définir un autre SecurityManager de permissivité compatible avec les contraintes de sécurité et de faisabilité pour ce navigateur. Ceci ne pourra être réalisé qu'en utilisant une applet signée (dans le cas contraire, il n'est pas possible de remplacer le SecurityManager standard).
Remarque : Lorsque l'applet est signée, il n'est pas toujours nécessaire de créer un nouveau SecurityManager pour modifier ses droits. Certains navigateurs permettent de modifier ces droits par simple configuration.

Justification

Sans Objet

Exemple

Sans Objet.

QA.SecurApplication	On doit sécuriser les applications Java standalone.
M=1;F=3;P=11;V=0	
Quelconque	

Description

Dans toute application Java " Standalone ", on doit définir et implémenter des règles de sécurité.

Justification

Java n'impose aucune contrainte de sécurité sur les applications " standalone ", c'est à dire les applications qui ne sont pas censées être lancées à partir d'un navigateur. Lorsque ces applications ne s'exécutent que sur une seule machine sans faire aucune connexion réseau, on peut considérer qu'a priori elles ne sont pas hostiles, du moins tant que l'utilisateur s'impose des règles de sécurité suffisantes.
Par contre, dès qu'une telle application doit se connecter à un serveur, les risques d'agression et d'infection existent au même titre que pour une applet dans un navigateur. Il faut donc dans ce cas définir des règles de sécurité réseau en amont de la réalisation proprement dite.

Exemple

On pourra par exemple se définir un SecurityManager spécialisé pour l'application, et adapter les ClassLoader et BytecodeVerifier standard. Mais attention, les appels au SecurityManager, dans le cas d'une application " Standalone ", doivent être faits explicitement par le programme, rien n'est pris en charge par la machine virtuelle Java dans ce cas, comme c'est le cas pour les applets.
On pourra protéger l'accès au jar ou aux classes de l'application pour empêcher toute falsification.
Remarque : l'utilisation d'un SecurityManager peut engendrer une perte importante des performances de l'application. Dans le cas où les performances sont critiques, on pourra implémenter manuellement la sécurité dans l'application en vue de faire certifier son application. Pour ce type d'application la sécurité se gère à un autre niveau par des ACL (Access Control List), de l'authentification, avec JCE...

QA.AppletTest	On doit toujours tester les applets dans un navigateur.
M=1;F=3;P=12;V=0	
Applet	

Description

Toute applet, en plus de tests standards de fonctionnement, par exemple avec un appletviewer, doit être testée également à l'intérieur d'un navigateur.

Justification

Les contraintes de sécurité appliquées aux applets dans le cas d'un chargement avec un appletviewer sont beaucoup moins restrictives que dans un navigateur. En particulier, les règles de manipulation des fichiers par les applets peuvent être très différentes : les appletviewers permettent souvent de manipuler des fichiers en local et sur le serveur (lecture et écriture).
Il est impératif de tester les applets en environnement réel, ne serait-ce que pour s'assurer en permanence qu'elles ne violent pas de règle de sécurité.

Exemple

En général, on interdit aux applets de manipuler des fichiers locaux et de se connecter à d'autres serveurs que celui d'origine, dans le cas d'une connexion à un serveur distant à l'intérieur d'un navigateur.
Cela peut néanmoins être nécessaire fonctionnellement.
On devra dans tous les cas s'assurer que toutes les contraintes de sécurité sont respectées.

QA.Obfuscator	Il faut encrypter le bytecode des applications ou des applets contenant du code " confidentiel ".
M=0;F=2;P=45;V=1	
Quelconque	

Description

Le bytecode des applications " sensibles " téléchargées sur le réseau, en particulier celui des applets, doit être rendu illisible ou incompréhensible, par exemple en utilisant un " obfuscator ".
Remarque : l'utilisation de ce type d'outils rendra très difficile, voire impossible l'utilisation des mécanismes dynamiques de découvertes (introspection) et d'invocation de méthodes du langage Java.

Justification

Le bytecode Java généré par un compilateur Java standard peut maintenant être facilement décompilé (car c'est un " assembleur " de haut niveau qui a été conçu pour être vérifiable par le Verifier). Toute personne ayant téléchargé une applet Java peut récupérer les bytecodes dans le cache du navigateur et désassembler le code avec javap (du JDK) ou le décompiler avec d'autres outils comme Jasmine ou Mocha. On a alors le code source des classes importées presque identique à l'original, en tout cas récupérable et modifiable à loisir…
Il existe cependant au moins un moyen de crypter partiellement les bytecodes : rendre la lecture du code incompréhensible en changeant les identificateurs du programme. L'algorithme lui-même reste lisible, mais extrêmement difficile à déchiffrer. Il existe des programmes qu'on appelle obfuscators qui parsent le code source et rendent illisibles et incompréhensibles les programmes (en théorie). Citons Zipper, HashJava, Crema, Macho par exemple.

Le seul moyen de cryptage complet du bytecode réside dans l'utilisation d'algorithmes de chiffrement. Ce cryptage peut être fait à deux niveaux :
au niveau des connexions réseau en utilisant SSL (Secure Socket Layer) ;
au niveau du bytecode : le bytecode doit être crypté avant d'être stocké puis décrypté par le ClassLoader au moment de son chargement. Toutes les technologies nécessaires pour réaliser ces cryptage/décryptage sont présentes dans le JDK.
On pourra dans les cas très sensibles compléter le travail des obfuscators en modifiant les algorithmes des méthodes pour les rendre moins compréhensibles, sans en abuser car ceci introduit souvent des bogues supplémentaires.

Exemple

Sans Objet.

QA.CLASSPATH	Il faut protéger particulièrement les packages standard Java.
M=0;F=2;P=15;V=0	
Quelconque	

Description

Il est indispensable de protéger l'accès aux classes standard de l'API Java.

Justification

Les classes de base de l'API Java et des browsers Web intégrant des machines virtuelles Java sont stockées dans des fichiers installés sur les postes clients. Ces fichiers contiennent notamment les classes ClassLoader et SecurityManager qui pourraient être modifiés (par un virus classique, i.e. non Java) pour permettre à des applets de violer les mécanismes de sécurité.
Il faut protéger (autant que l'on peut le faire) ces classes de bases par des mécanismes classiques de protection des fichiers (droit d'écriture) ou des mécanismes supplémentaires de vérification des classes utilisées. Rendre les bytecodes incompréhensibles (par l'homme) et ajouter des systèmes de vérification supplémentaires dans l'application elle-même rendra la tâche du client malveillant plus difficile.
On peut noter que les classes de base de Netscape Communicator 4.5 sont signées : elles sont donc protégées contre les attaques évoquées plus haut puisque cette signature garantie leur intégrité.

Exemple

Inclure un mécanisme interne de vérification de version ou de " checksum " sur les bytecodes, si possible disséminé dans plusieurs classes pour protéger un peu plus le code de l'application et le serveur lui-même.

QA.JavaFlaws	Il faut connaître les problèmes de sécurité des outils Java utilisés.
M=1;F=3;P=25;V=0	
Quelconque	

Description

On doit toujours se tenir informé des problèmes de sécurité existant dans les outils Java utilisés.

Justification

Bien que Java dispose de modèles et mécanismes de sécurité parmi les meilleurs qui soient, il existe pour chaque version d'outil Java utilisé un certain nombre de lacunes. On peut espérer à terme disposer de mécanismes de sécurité presque inviolables, mais ce n'est pas le cas actuellement, essentiellement en ce qui concerne les clients malveillants.
Pour pouvoir parer à toute éventualité d'attaque des serveurs, il faut connaître ces lacunes de sécurité concernant les outils utilisés.

Exemple

Visiter régulièrement les sites Internet suivants : http://java.sun.com/security, http://www.rstcorp.com/javasecurity/links.html

QA.OptimJAVA	Il faut utiliser l'option de compilation optimisée de Java.
M=0;F=0;P=25;V=2	
Quelconque	

Description

L'option de compilation ' -o ' du compilateur Java doit être utilisée, au moins en version d'exploitation.

Justification

Cette option de compilation permet au compilateur de remplacer les expressions calculables par des constantes, de gérer des bytecodes plus efficace pour les boucles, de supprimer dans une moindre mesure le code mort comme if (false) i = 1. Ces optimisations sont minimales, en tout cas en regard des compilateurs FORTRAN ou C modernes.
Cependant les versions successives des compilateurs s'améliorent sans cesse, et il vaut mieux prendre immédiatement l'habitude d'utiliser cette option, qui déjà permet une amélioration minimale des bytecodes Java générés.

Exemple

```
java -o Myclass.java
```

QA.OptimStrings	Si une optimisation des performances sur la gestion des strings est nécessaire, il faut utiliser la méthode `append` plutôt que l'opérateur +.
M=0;F=0;P=90;V=1	
Quelconque	

Description

Le langage Java fournit la classe String, assurant une gestion simple et pratique des chaînes de caractères par les programmeurs. En particulier, on peut réaliser la concaténation de deux strings avec l'opérateur + de la classe String au lieu d'avoir à utiliser la méthode append de la classe StringBuffer.
Dans le cas où une optimisation est nécessaire, on préférera cependant la méthode append fournie par l'API à l'opérateur + du langage. Ceci est également vrai pour l'opérateur +=.

Justification

Les objets de la classe String sont des invariants : une fois instanciés, on ne peut plus les modifier. Pour réaliser la concaténation de deux chaînes de caractères en utilisant l'opérateur +, le compilateur Java passe par l'utilisation d'instances de la classe StringBuffer, qui elles ne sont pas invariables.

Exemple

```
String s = "toto";
```

est en réalité traduit par :

```
String s = new StringBuffer().append("toto").toString();
```

le tableau de caractères toto étant passé en paramètre à la méthode append. Lors de l'utilisation de l'opérateur +, le compilateur crée une instance StringBuffer et concatène les deux tableaux de caractères de par et d'autres de l'opérateur.

Par exemple :

```
String s = s1 + "toto";
```

se traduit en fait par :

```
String s = new StringBuffer().append(s1.toCharArray()).
                              append("toto").toString();
```

Cette particularité du compilateur simplifie le travail du programmeur, mais on voit ici que le compilateur crée non seulement un objet intermédiaire (un StringBuffer), ce qui est coûteux en soi, mais doit de plus convertir la chaîne s1 en tableau de caractères. De ce fait les expressions concaténant des chaînes de caractères à l'aide de l'opérateur + sont peu efficaces.

On préférera donc utiliser judicieusement la classe StringBuffer et sa méthode append à l'opérateur +.

Exemple non optimisé

```
String chaine = "hello";
for (int i = 0; i < 10000; i++) {
    chaine += ?World?;
}
```

Exemple optimisé

```
StringBuffer chaine = new StringBuffer(?hello?);
for (int i = 0; i < 10000; i++) {
    chaine.append(?World?);
}
```

Les benchmarks mettent en évidence un temps d'exécution supérieur d'un facteur proche de 100 pour l'opérateur +.

QA.Synchro	Il faut n'utiliser le mot-clef `synchronized` que si nécessaire.
M=0;F=0;P=78;V=1	
Quelconque	

Description

Le mot-clef synchronized en Java sert à empêcher des threads d'accéder simultanément aux mêmes données. Il peut être appliqué à des méthodes ou à des blocs de code.

On n'utilisera pas ce mot-clef de manière abusive pour essayer de garantir que le code est " thread-safe ".

Justification

La synchronisation est extrêmement coûteuse. Chaque synchronisation entraîne un surcroît de contrôles, d'appels de méthodes, et de changements de contexte qui dégrade considérablement les performances du programme.
Les benchmarks référencés rapportent un facteur de ralentissement de 10 à 50 si l'on marque une méthode comme synchronisée. Pour une meilleure efficacité, il ne faudra donc marquer des méthodes comme synchronisées uniquement si nécessaire.
Attention : il faut toutefois veiller à synchroniser toutes les sections critiques dans les programmes multi-thread, sous peine d'avoir un programme ne s'exécutant pas correctement.

Exemple

Exemple incorrect

```
public synchronized float calculerMoyenne() {
    int nbElems = valeurs.length;
    for (int i = 0; i < nbElems; i++) {
        moyenne += valeurs[i];
    }
    return (float) moyenne / (float) nbElems;
}
```

Exemple correct

```
public float calculerMoyenne() {
    int nbElems = valeurs.length;
    // Cette methode ne change pas l'etat du recepteur :
    // Les données sont en lecture seulement.
    for (int i = 0; i < nbElems; i++) {
        moyenne += valeurs[i];
    }
    return (float) moyenne / (float) nbElems;
}
```

QA.Access	Il faut limiter l'implémentation des accesseurs (get/set/is) au minimum.
M=1;F=0;P=95;V=1	
Quelconque	

Description

Tous les attributs d'une classe n'ont pas besoin d'accesseurs. Il ne faut pas présumer de l'usage d'un accesseur et surcharger inutilement le code et la documentation d'une classe. Les méthodes d'accès aux attributs privés d'une classe devront donc être définies de manière à être suffisantes pour l'usage préconisé. On prendra toutefois garde à implémenter suffisamment d'accesseurs pour assurer la réutilisation de la classe.
Attention, certains outils de conception (comme Rational Rose) peuvent être paramétrés pour générer automatiquement des méthodes d'accès pour tous les attributs d'un objet.

Justification

La surcharge de travail liée à la création et à la maintenance d'accesseurs non nécessaires nuit à la productivité.
Les variables de classes sont souvent dépendantes les unes des autres et le fait d'autoriser leur lecture et écriture indépendamment complexifie (souvent inutilement) le code. De plus, l'implémentation de

méthodes n'étant pas utilisées dans le cadre du projet augmente fortement les risques de bug lors d'une utilisation ultérieure de ces dernières.

Exemple

```
Public class Maliste {
    private int taille;
    private double[] elements;

    // Cette classe implementera les methodes d'acces suivantes :
    //    public int      getTaille()
    //    public void     ajoutElement(double valeur)
    //    public void     supprimeElement(int numero)
    //    public void     definirElement(int numero, double valeur)
    //    public double[] getElements()

    // Les accesseurs suivants ne seront pas implante :
    //    void setTaille(int taille)
    //    void setElements(double[] elements)
    ...
}
```

QA.PackBase	Une connaissance correcte des packages `java.lang, java.util` et `java.io` est indispensable dès la conception.
M=3;F=1;P=10;V=0	
Quelconque	

Description

Les packages java.* font partie intégrante du langage Java.
Le package java.lang est une extension immédiate du langage ; il définit par exemple les classes Integer, Float, ... qui permettent de gérer les valeurs numériques comme des objets en fournissant les services de conversion etc. associés.
Le package java.util contient des classes utilitaires dont il est difficile de se passer : String, StringBuffer, StringTokenizer pour gérer les chaînes de caractères, HashMap et Dictionary pour gérer les collections,...
Le package java.util.zip contient des utilitaires gérant la compression/décompression de fichiers.
Le package java.io contient toute une série de classes permettant de gérer les entrées/sorties.
Il faut connaître et utiliser le contenu de ces librairies dès la phase de conception.

Justification

Cela permet en effet en phase de conception de définir de façon efficace des classes utilitaires spécifiques au projet en utilisant au mieux les fonctionnalités préexistantes.
En phase de développement cela permet d'écrire du code plus efficace et plus synthétique, ce qui facilite la maintenance et l'évolutivité du logiciel.

Exemple

Sans Objet.

QA.PackSpecif	Il faut utiliser les packages standard adaptés au type d'application réalisée.
M=3;F=1;P=11;V=0	
Quelconque	

Description

Suivant le type d'application réalisée, il faut utiliser dès la conception les packages fournis avec le langage. Il est nécessaire de bien connaître le contenu des packages du JDK dès la phase de conception afin de les utiliser au plus tôt s'ils sont adaptés aux besoins de l'application.
On trouvera la liste exhaustive de ces packages dans la documentation HTML du JDK (page : api/overview-summary.html).
En outre, on peut étendre ces packages standard aux packages standard « de fait » (ex. Fondation Apache).

Justification

Même justifications que pour la règle QA.PackBase.

Exemple

Quelques packages utiles :

Package	Usage
java.lang	Doit toujours être utilisé, il contient les classes fondamentales de Java.
java.math	Doit être utilisé pour l'arithmétique de précision.
java.text	Doit être utilisé si l'application gère l'internationalisation des textes.
java.swing et ses nombreux sous-packages	Doivent être utilisés si l'application contient une Interface Homme Machine.
javax.swing.event java.awt.event	Doivent être utilisés pour gérer les événements graphiques.
java.applet	Doit être utilisé en complément de la classe javax.swing.JApplet si l'application est activée depuis une page HTML accessible depuis un navigateur.
java.net	Doit être utilisé si l'application manipule des sockets ou des URLs.
java.awt.image	Doit être utilisé si l'application gère des images.
java.awt.datatransfer	Doit être utilisé si l'on envisage de transférer des données entre applications (sur le mode copier-coller).
java.beans	Doit être utilisé pour définir des JavaBeans.
java.lang.reflect	Permet à un programme d'accéder aux fonctionnalités d'introspection disponibles dans le langage Java (découverte dynamique du contenu d'une classe et instanciation/invocation dynamique de méthode).
java.rmi org.omg ainsi que leurs sous-packages	Peuvent être utilisés pour créer des objets distribués et accéder à des objets distants.
java.security	Doivent être utilisés si on gère la sécurité de façon explicite

et ses sous-packages	(cryptage, signature électronique des classes, sécurité introduite dans une application standalone).
java.sql	Doit être utilisé de préférence pour s'interfacer avec une base de données (il faut connaître son contenu avant d'envisager l'utilisation du package propriétaire d'une base de donnée spécifique).

QA.AdaptExt	Il faut adapter le type d'interface avec l'extérieur aux besoins du projet
M=3;F=0;P=23;V=0	
Quelconque	

Description

Il faut structurer l'application ou l'applet en tenant compte des besoins réels en termes de distribution ; on peut distinguer trois cas de figure allant de la solution la moins distribuée à la solution la plus distribuée :
Chargement de fonctions natives par l'intermédiaire d'une librairie dynamique.
Il y a alors un seul exécutable (multi-threadé) : l'interpréteur Java.
Appel d'exécutables extérieurs à l'aide de la méthode System.exec(...).
Il y a alors exécution d'un autre programme sur le même ordinateur.
C'est la méthode la plus simple et la plus efficace dans le cas d'un contexte mono-machine ; elle présente l'avantage de découpler le code Java du code natif ce qui permet de conserver les qualités de robustesse du programme Java, surtout au niveau de la gestion de la mémoire.
Cette solution est moins coûteuse en termes d'effort de développement que la précédente.
Appel à des services distants de façon plus ou moins élaborée :
connexion directe par sockets ou URL (package java.net),
utilisation de l'API Java RMI (Remote Method Invocation),
utilisation d'un middleware CORBA
Ces solutions sont les plus souples en termes de distribution de l'application, mais nécessitent bien sûr un effort de développement supplémentaire.

Justification

Dans le cas où l'on est obligé de faire appel à des ressources extérieures, il faut bien évaluer le besoin et choisir une solution adaptée de façon à limiter les coûts de développement et de maintenance.

Exemple

Sans Objet.

7.10. AUTRES REGLES

Beans.Choix	Il convient de choisir les composants que l'on désire transformer en JavaBeans.
M=2;F=0;P=60;V=0	
Quelconque	

Description

Il faut définir des règles permettant de déterminer les composants susceptibles de devenir des JavaBeans.
On pourra par exemple se poser les trois questions suivantes :
Le composant est-il prévu pour être réutilisé ?
Le composant doit-il être utilisé avec d'autres composants réutilisables ?
Vais-je utiliser un outil de développement IDE ?

Justification

Le développement de JavaBeans entraîne un surcoût et une surcharge du code (si aucun outil n'est utilisé) qu'il ne faut pas généraliser à tous les composants d'un projet. On prendra tout de même soin grâce à l'application des règles de nommage de faciliter la transformation ultérieure de tous les objets.

Exemple

Sans Objet.

Beans.Transient	Toutes les données dont la sauvegarde n'est pas nécessaire doivent être déclarées `transient`.
M=1;F=0;P=62;V=1	
Quelconque	

Description

La sérialisation Java permet aux objets implémentant l'interface Serializable d'être transformés en une séquence binaire pouvant être sauvegardée et restaurée de manière à conserver un objet dans son état courant. Les types de base Java ainsi que les classes du JDK sont Serializable. Dans le cadre des JavaBeans, la sérialisation est appliquée par défaut à toutes les données non statiques.
Lors de la création de composants JavaBeans, on doit définir par défaut tous les éléments comme transient. Seuls les éléments à sauvegarder devront être sérialisables.
On prêtera une attention particulière aux listeners des évènements générés par le composant en déclarant transient les vecteurs de listeners. La sérialisation d'un vecteur de listeners pourrait entraîner la sauvegarde de l'arborescence d'objets associés aux évènements d'un Bean de plus, la gestion des relations entre composants (et donc des listeners) ne doit être réalisée que par les composants de type containers.

Justification

La sérialisation automatique peut ainsi entraîner la sauvegarde de beaucoup d'informations superflues. Il convient donc d'identifier correctement les éléments strictement nécessaires à la restauration d'un Bean.

Exemple

Sans Objet.

Beans.Notification	Un composant IHM « JavaBean » doit générer un événement lié à tout changement significatif de son aspect.
M=1;F=1;P=67;V=1	
Quelconque	

Description

Les évènements générés par un Bean doivent prendre en compte deux points de vue :
ils doivent être adaptés à l'usage des auditeurs potentiels, tous les changements significatifs doivent être notifiés.

Justification

Les composants " encapsulant " un Bean doivent pouvoir prendre en compte les modifications du Bean (notamment les changements d'aspect).
Il faut toutefois faire très attention à ne pas créer des composants " bruyants " (générant trop d'évènements). Il faut donc éviter de regrouper toutes les modifications d'un composant dans un même événement (pour ne pas notifier inutilement tous les auditeurs et ainsi risquer de fortes baisses de performances).

Exemple

Si un composant représente une valeur numérique sous forme d'un champ de texte, d'un ascenseur et d'un graphique, il peut-être judicieux de ne générer qu'un seul événement lié à la modification de l'un des trois éléments.
Par contre, si ces trois mêmes éléments graphiques représentent 3 valeurs différentes, on utilisera de préférence 3 évènements distincts (tout en prenant soin de vérifier que l'usage de ces évènements est bien lié à 3 types d'auditeurs distincts).

Applet.Applications	Il faut fournir les applets également sous forme d'application.
M=2;F=1;P=70;V=2	
Applet	

Description

On doit fournir une méthode main dans toute applet de manière à pouvoir la faire fonctionner également comme application standalone.

Justification

Java permet de manipuler les applets en tant qu'application de façon aisée (voir exemple ci-dessous). Ceci permet alors un double fonctionnement des applets, en remarquant toutefois que l'inverse n'est pas vrai : toute application ne peut être transformée en applet, ne serait ce que pour des raisons de sécurité.
On remarquera de plus que le déboguage des applications dans des outils IDE ou avec le JDK standard de Sun est plus facile qu'avec des applets, d'où l'intérêt de disposer en permanence d'une version " applicative " de l'applet. Enfin on disposera d'un meilleur contrôle du code en phase de développement.
Ceci ne devra pas cependant faire oublier les règles élémentaires et indispensables de sécurité concernant les applets. L'intérêt de cette manière d'opérer dans le cas général est de pouvoir développer l'applet avec toute la liberté et la puissance de développement possible.

Exemple

```
import java.applet.*;
import java.awt.*;

public class StandaloneApplet extends Applet {
    public static void main(String args[]) {
        Applet applet = new StandaloneApplet();
        Frame frame = new AppletFrame("myApplet", applet, 300, 300);
    }
}

class AppletFrame extends Frame {
  public AppletFrame(String title, Applet applet, int width, int height)
{
    super(title);

    // Ajout du code spécifique à la fenêtre utilisateur.
    ...
    this.resize(width, height);
    this.show();

    // Démarrage de l'applet.
    applet.init();
    applet.start();
}
```

On remarquera ici le découpage de la partie fabrication de fenêtre qui permet l'affichage de l'applet en tant qu'application (classe AppletFrame). Cette classe démarre l'applet comme le ferait un appletviewer avec bien moins de possibilité cependant (la classe AppletContext n'est pas prise en compte dans cet exemple et toute méthode afférente sera inutilisable). On pourra étendre cette manière de travailler en construisant une classe AppletViewer personnalisée imposant seulement les contraintes de sécurité désirées pour les applications standalone.

Applet.StartStop	On doit démarrer et arrêter les tâches de fond d'une applet dans les méthodes `start()` `stop()` respectivement.
M=1;F=1;P=72;V=1	
Applet	

Description

Toute tâche de fond d'une applet (animation, activité permanente) doit être démarrée dans la méthode start() de l'applet et arrêtée dans la méthode stop() de l'applet.
Attention cependant au comportement de ces méthodes dans certains navigateurs. Elles peuvent être déclenchées dans d'autres cas que lorsque le navigateur quitte/revient sur l'applet (cf. Netscape qui appelle ces méthodes lorsque la fenêtre du navigateur est retaillée).
On prendra toutes les précautions nécessaires pour obtenir des fonctionnements similaires quel que soit le navigateur utilisé au niveau des méthodes start(), stop(), init(), …
Remarque : des informations sur ce sujet peuvent être trouvées sur le site JavaWorld.

Justification

Les méthodes init et destroy d'une applet sont appelées à la création et à la destruction de l'applet par l'appletviewer ou le navigateur la visualisant. Toute tâche de fond qui est active pendant toute la durée de vie de l'applet doit commencer et finir dans ces deux méthodes
Par contre, dans beaucoup de cas, on voudrait que ces tâches de fond (comme les animations par exemple) ne soient actives que quand l'applet est visible. Dans ce cas, pour diminuer la charge du processeur et pour une meilleure architecture, on démarrera ces activités uniquement dans les méthodes start et stop qui sont activées respectivement quand l'applet devient visible ou invisible à l'utilisateur.

Exemple

Dans le cas des applets fournissant une petite animation, on spécifiera que l'applet implémente l'interface Runnable et on démarrera dans la méthode start() de l'applet une instance de Thread pointant sur l'applet instanciée, grâce à l'appel Thread.start(). Ainsi la méthode run() de l'applet sera appelée par ce thread, découplant de ce fait la mécanique de fonctionnement des applets dans un navigateur des contrôles et calculs nécessaires à l'animation. On appellera la méthode Thread.stop() pour stopper ce thread d'animation dans la méthode stop() de l'applet.

Applet.Animation	Il faut utiliser des threads dédiés pour réaliser des animations dans des applets.
M=1;F=2;P=65;V=1	
Applet	

Description

Toute tâche de fond d'une applet (animation, activité permanente) doit être prise en charge par un thread dédié à cette activité.

Justification

C'est nécessaire pour éviter le blocage permanent de l'application sur la tâche de fond.
Cela permet aussi de synchroniser le passage de données inter-applets au sein de leur navigateur ou appletviewer.

Exemple

```
public class Animation extends Applet implements Runnable {
    // Thread de support d'animation, avec redefinition
    // des methodes start et stop pour démarrer et arrêter.
    private Thread animatorThread = null;
    ...
    public void init() {
        // Instanciation et initialisation de l'applet.
    }
    public void start() {
        if (animatorThread == null) {
            animatorThread = new Thread(this);
            animatorThread.start();
        }
    }
    public void stop() {
        if ((animatorThread != null) &&
             animatorThread.isAlive()) {
            animatorThread.stop();
```

```
                animatorThread = null;
            }
        }

        // Lancement de l'animation.
        public void run() {
            while(true) {
            ....
            try { Thread.sleep(200); }
                catch (InterruptedException e) {; }
            }
        }
    }
```

Applet.Debugging	Il faut déboguer les fonctionnalités d'une applet dans un appletviewer plutôt que dans un navigateur.
M=2;F=2;P=18;V=0	
Applet	

Description

En phase de développement ou d'évolution d'une applet, on s'efforcera de la faire fonctionner à l'aide d'un appletviewer plutôt que d'un navigateur.

Justification

Les appletviewers permettent de visualiser des applets contenus dans un fichier HTML, et seulement les applets. Lorsque plusieurs applets sont référencés dans le fichier, l'appletviewer ouvre une fenêtre d'applet pour chacune d'entre elles. Le développeur se préoccupe donc uniquement du bon fonctionnement de l'applet et pas des aspects ergonomiques de son insertion dans la page HTML. Les navigateurs affichent quant à eux tous les composants visualisables du fichier HTML. Ceci peut ralentir et perturber la productivité du développeur, d'autant qu'à chaque nouvelle version d'applet il faut redémarrer un nouveau navigateur.
L'utilisation d'un appletviewer permet de tirer partie du relâchement des contraintes de sécurité imposées par les navigateurs, ce qui autorise l'utilisation de mécanismes de déboguage via jdb ou par lecture-écriture de fichiers sur la machine locale, ou le profiling., ou de spécifier des paramètres particuliers à l'interpréteur Java.
On utilisera par contre régulièrement les navigateurs pour visualiser les applets en cours de développement, lorsque l'on voudra " faire le point " sur l'état courant de l'applet ou un test en vraie grandeur, lorsque l'on voudra tester ponctuellement les aspects sécurité, lorsque l'on sera en phase finale de développement.

Exemple

Sans Objet.

Applet.TagName	Il faut toujours renseigner le paramètre `NAME` de la balise `APPLET` des fichiers html contenant une applet Java.
M=1;F=0;P=35;V=2	
Applet	

Description

Ce paramètre NAME désigne l'applet chargée par un nom.

Justification

Le paramètre NAME de la balise APPLET permet de désigner nommément une applet à l'intérieur d'une page HTML visualisée par un navigateur ou un appletviewer. Elle permet au navigateur, lors des communications inter-applets dans une page, de retrouver une applet visualisée en appelant la méthode getApplet() de la classe AppletContext.
La balise OBJECT est conseillée dans les dernières versions d'HTML à la place d'APPLET. Néanmoins ce tag est encore très utilisé (y compris dans les exemples du dernier JDK).

Exemple

```
<applet codebase="http://my.domain.fr/"
    code="NervousText.class" width=400 height=75
    name = ?monApplet?>
</applet>
```

Applet.TagParam	Il faut utiliser la balise `PARAM` pour paramétrer les applets.
M=1;F=0;P=36;V=2	
Applet	

Description

Dans les fichiers HTML chargeant des applets Java, on s'efforcera de paramétrer le plus possible les applets avec les variables spécifiées par la balise PARAM.

Justification

La balise PARAM permet de donner une valeur à un paramètre nommé de l'application. Moins pratique que les arguments d'une ligne de commande, cette balise permet cependant de configurer le comportement d'une applet par l'intermédiaire du fichier HTML le contenant. A charge au serveur suivant les cas de donner une valeur différente à ce paramètre.

Exemple

```
<applet codebase="http://my.domain.fr/applets"
    code="monApplet.class" width=400 height=75>
    <param name="niveau" value="expert">
</applet>
```

Applet.MultiSynchro	Ne pas s'appuyer sur un ordre de chargement a priori des différentes applets contenues dans une page HTML.
M=1;F=2;P=32;V=0	
Applet	

Description

Lorsqu'un fichier HTML faisant référence à plusieurs applets Java est chargé, on ne peut pas faire d'hypothèse sur l'ordre dans lequel les applets sont chargées, et notamment sur l'ordre dans lequel leurs méthodes init seront appelées.
Si on souhaite que les applets communiquent entre elles, on fera en sorte qu'un mécanisme de synchronisation charge les applets dans un certain ordre ou en assure le fonctionnement synchronisé.
Ce mécanisme de synchronisation doit être défini dans le code Java et n'existe pas pour l'instant en standard.

Justification

Le chargement des applets et des classes utilisées par celles-ci est effectué de manière asynchrone par les navigateurs. Ainsi on ne peut prévoir d'avance quelle applet sera la première chargée et active, et on ne peut donc pas s'appuyer sur un ordonnancement de chargement à priori. D'autre part des applets chargées par des pages HTML différentes (à l'aide de " frame " par exemple) peuvent être amenées à communiquer ensemble.
Pour réaliser cette communication, il est recommandé d'implémenter des mécanismes qui en assurent le bon fonctionnement, par exemple en s'assurant de l'existence et de la disponibilité de l'applet distant.

Exemple

Si les applets sont dans une même page HTML, on pourra s'appuyer sur l'AppletContext fourni par le browser. On l'interrogera (getApplet(name) ou getApplets()) pour récupérer une référence à l'applet distante, qu'on pourra ensuite interroger au travers d'appels de méthodes.
Si les applets sont situées dans des pages HTML différentes, on pourra utiliser des variables statiques d'une classe commune permettant de savoir quelles sont les applets actuellement chargées et disponibles pour communiquer.

Applet.Jar	Il faut regrouper tous les composants d'une applet dans un fichier " .jar " unique.
M=2;F=1;P=67;V=2	
Applet	

Description

Tous les éléments utilisés par une applet (aussi bien les classes que les images ou les sons), seront placés dans un même JAR (Java Archive).

Justification

Une applet peut prendre beaucoup de temps à se charger car elle doit télécharger tous ses composants à chaque fois en utilisant une connexion serveur distincte (le cache du browser peut être utilisé mais ce n'est pas garanti).
Un fichier JAR unique contenant tous les éléments d'une applet pourra être téléchargé en une seule fois et la signature des composants pourra tout de même être utilisée individuellement pour chaque entrée du fichier JAR.

Exemple

Sans Objet.

8. AUTRES ASPECTS SPECIFIQUES AU LANGAGE

Sans objet

9. SYNTHESE

9.1. TABLE RECAPITULATIVE DES REGLES

Les règles sont récapitulées ici, classées par ordre alphabétique.

Id. Règle	Intitulé	Page
Applet.Animation	Il faut utiliser des threads dédiés pour réaliser des animations dans des applets.	70
Applet.Animation	Il faut expliciter les classes importées.	13
Applet.Applications	Il faut fournir les applets également sous forme d'application.	68
Applet.Debugging	Il faut déboguer les fonctionnalités d'une applet dans un appletviewer plutôt que dans un navigateur.	71
Applet.Jar	Il faut regrouper tous les composants d'une applet dans un fichier " .jar" unique.	73
Applet.MultiSynchro	Ne pas s'appuyer sur un ordre de chargement a priori des différentes applets contenues dans une page HTML.	72
Applet.StartStop	On doit démarrer et arrêter les tâches de fond d'une applet dans les méthodes `start()` `stop()` respectivement.	69
Applet.TagName	Il faut toujours renseigner le paramètre `NAME` de la balise `APPLET` des fichiers html contenant une applet Java.	71
Applet.TagParam	Il faut utiliser la balise `PARAM` pour paramétrer les applets.	72
Beans.Choix	Il convient de choisir les composants que l'on désire transformer en JavaBeans.	67
Beans.Notification	Un composant IHM « JavaBean » doit générer un événement lié à tout changement significatif de son aspect.	68
Beans.Transient	Toutes les données dont la sauvegarde n'est pas nécessaire doivent être déclarées `transient.`	67
Don.AllocNull	Il est inutile de tester l'allocation des objets.	27
Don.BitSet	L'utilisation de la classe `BitSet` est interdite.	30
Don.Dictionnaire	Il faut éviter d'utiliser la classe `Hashtable` et choisir le type de classe " données indexées " en fonction de l'application visée.	29
Don.Ensemble	Il faut adapter le type de classe " ensemble " implémenté en fonction de l'application visée.	28
Don.Inner	On ne doit pas abuser des classes incluses non statiques.	24
Don.Interface	La notion d'interface Java doit être utilisée à bon escient.	21
Don.InterfaceEstUn	La notion d'interface doit implémenter une relation conceptuelle " est un " ou la relation conceptuelle " est une implémentation de ".	19
Don.IntInstance	Une classe de base dont on veut interdire l'instanciation doit être définie comme une classe abstraite.	18
Don.List	On ne doit pas utiliser la classe Vector et choisir le type de classe "liste" le plus adapté en fonction de l'algorithme.	27
Don.MembreStatic	On ne doit pas abuser des classes ou interfaces membres statiques.	23
Don.MembreVis	Il est interdit de modifier la visibilité des classes membres.	26
Don.MéthAbstr	Il faut préférer les méthodes abstraites aux méthodes de corps vide.	22
Don.TableauxDecl	Il faut déclarer les tableaux de façon uniforme.	26
Dyn.Daemon	Les threads effectuant des traitements de fond doivent être définis comme des " démons ".	44
Dyn.Exit	Il faut fermer les threads non " démons " avant d'appeler `System.exit().`	44
Dyn.TempsRéel	Il faut adapter le type d'interface avec l'extérieur aux besoins du projetIl ne faut pas utiliser Java pour les applications à forte contrainte temps réel.	45
Dyn.ThreadCreation	Il faut utiliser l'interface `java.lang.Runnable` pour créer un thread.	43

Id. Règle	Intitulé	Page
QA.SystemCall	Il ne faut pas appeler de méthode ouverte au système.	49
QA.TestUnitaire	Il faut introduire dans les classes développées les méthodes `main()` et `toString()`.	46
QA.VersionNavig	Il faut s'assurer que la version des navigateurs utilisés supporte le JDK utilisé.	47
Tr.AffectMask	Aucune affectation ne doit être masquée par une autre opération.	33
Tr.Blocs	Tous les blocs d'une instruction de contrôle doivent être délimités par des accolades.	38
Tr.ConstrBut	Il faut réserver les constructeurs aux tâches d'initialisation.	30
Tr.ConstRetour	Il ne faut pas définir de " constructeur " avec une valeur de retour.	31
Tr.DestrFinalize	Il ne faut pas redéfinir la méthode `finalize()`.	31
Tr.InstanceOf	Il faut utiliser l'opérateur `instanceOf` avec parcimonie.	38
Tr.Iterations	Il faut réaliser les itérations avec l'interface `Iterator`.	39
Tr.MéthodeFinal	Il faut préfixer du mot clé `final` les méthodes dont on veut interdire la redéfinition.	35
Tr.MéthodeProtected	Une méthode ne doit être `protected` que si elle doit être utilisée depuis au moins une méthode définie dans une autre classe de la sous-hiérarchie.	34
Tr.MéthodePublic	Une méthode ne doit être `public` que si elle doit être utilisée depuis au moins une classe définie dans un autre package.	34
Tr.MéthodeRedef	Les définitions d'une fonction membre redéfinie entre une classe de base et une classe dérivée doivent comporter les mêmes noms de paramètres.	34
Tr.MéthodeSurcharge	Les différentes surcharges d'une fonction doivent offrir des services ayant la même sémantique.	35
Tr.RazReferences	Il faut remettre à `null` les références aux objets volumineux comme les tableaux.	32
Tr.SuperFinalize	En cas de redéfinition de la méthode `finalize()` on doit appeler `super.finalize()`.	32
Tr.SwitchBreak	Tout branchement d'un choix multiple doit se terminer par l'instruction `break`.	37
Tr.This	Il faut utiliser la référence `this` à l'objet courant avec parcimonie.	36

9.2. TRAÇABILITE VIS A VIS DU DOCUMENT « COMMUN »

Cette table a été créée lors de la mise en commun des règles entre langages. Elle donne pour chaque règle :

- L'ancienne identification de la règle (Version 4)
- La nature de la modification : Aucune, Renommée (on a changé l'identification de la règle), Communalisé (la règle a été déplacée vers les document Tout Langage), Supprimée.
- La nouvelle identification dans le cas où la règle n'a pas été supprimée, une raison de la suppression sinon.

Identification Version 4	Nature	Nouvelle identification
APPLET-Animation	Renommée	Applet.Animation
APPLET-AppletAppli	Renommée	Applet.Applications
APPLET-Debugging	Renommée	Applet.Debugging
APPLET-MultiSynchro	Renommée	Applet.MultiSynchro
APPLET-StartStop	Renommée	Applet.StartStop
APPLET-TagName	Renommée	Applet.TagName
APPLET-TagParam	Renommée	Applet.TagParam
BEANS-Choix	Renommée	Beans.Choix
BEANS-Notification	Renommée	Beans.Notification
BEANS-Transient	Renommée	Beans.Transient
CLASSE-Inner	Renommée	Don.Inner
CLASSE-InstanceOf	Renommée	Tr.InstanceOf
CLASSE-MembreStatic	Renommée	Don.MembreStatic
CLASSE-MembreVis	Renommée	Don.MembreVis
CLASSE-MéthAbstr	Renommée	Don.MéthAbstr
CLASSE-ProtégerDon	Commonalisée	Org.Masquage
CLASSE-This	Renommée	Tr.This
CONCEP-Exterieur	Renommée	QA.AdaptExt
CONCEP-Interface	Renommée	Don.Interface
CONCEP-InterfEstUn	Renommée	Don.InterfaceEstUn
CONCEP-IntInstance	Renommée	Don.IntInstance
CONCEP-Ressources	Renommée	Int.AccesRessource
CONSTR-But	Renommée	Tr.ConstrBut
CONSTR-Retour	Renommée	Tr.ConstRetour
CONTR-Break	Renommée	Tr.SwitchBreak
CONTR-ConditionFor	Commonalisée	Tr.ModifCondSortie
CONTR-Default	Commonalisée	Tr.OrdreChoix
CONTR-ParamFor	Commonalisée	Tr.ModifCompteur
DESTR-Finalize	Renommée	Tr.DestrFinalize
DESTR-SuperFinalize	Renommée	Tr.SuperFinalize
DOC-Détail	Renommée	Pr.JavaDocDetail
DOC-Dev	Renommée	Pr.CommentDev
DOC-JavaDoc	Renommée	Pr.JavaDoc
DOC-MiseEnPage	Commonalisée	Pr.Aeration;Pr.CartStd;Pr.CartDonnée;Pr.CommIdent
DOC-Version	Renommée	Pr.GestConf
EXCEP-CatchUtil	Commonalisée	Err.Operation
EXCEP-Error	Renommée	Err.ClassError
EXCEP-RuntimeExcep	Renommée	Err.RuntimeExcep

Identification Version 4	Nature	Nouvelle identification
EXCEP-SousClasse	Renommée	Err.SousClasException
EXCEP-Specifique	Renommée	Err.ExcepSpecific
FORMA-PackBase	Renommée	QA.PackBase
FORMA-PackMetier	Supprimée	Trop générale
FORMA-PackSpecif	Renommée	QA.PackSpecif
MAINT-AffectMask	Renommée	Tr.AffectMask
MAINT-Const	Commonalisée	Don.Enumeration
MAINT-Debug	Renommée	QA.TestUnitaire
MAINT-Outil	Supprimée	Régle d'utilisation d'outil
MEM-RazRéférence	Renommée	Tr.RazReferences
MEM-TempsRéel	Renommée	Dyn.TempsRéel
METH-Final	Renommée	Tr.MéthodeFinal
METH-Protected	Renommée	Tr.MéthodeProtected
METH-Public	Renommée	Tr.MéthodePublic
METH-Redéfinie	Renommée	Tr.MéthodeRedef
METH-Retour	Commonalisée	Err.Mecanisme
METH-Surcharge	Renommée	Tr.MéthodeSurcharge
NOM-AccèsAttribut	Commonalisée	Id.Procedure;Id.Fonction
NOM-Classe	Commonalisée	Id.ClasseType
NOM-Constante	Commonalisée	Id.ConstSignif
NOM-Défaut	Commonalisée	Id.IdentRegle
NOM-Explicite	Commonalisée	Id.IdentSignif
NOM-Package	Renommée	Id.Package
NOM-Widget	Renommée	Id.Widget
OPTIM-9010	Commonalisée	Qa.Performances
OPTIM-Access	Renommée	QA.Access
OPTIM-AccesVars	Commonalisée	Don.Localite
OPTIM-AlgoStructs	Suprimée	Trop générale
OPTIM-AppletJar	Renommée	Applet.Jar
OPTIM-BitSet	Renommée	Don.BitSet
OPTIM-Compilation	Renommée	QA.OptimJAVA
OPTIM-InvBoucle	Commonalisée	Qa.Factorisation
OPTIM-Iteration	Renommée	Tr.Iterations
OPTIM-List	Renommée	Don.List
OPTIM-Map	Renommée	Don.Dictionnaire
OPTIM-Null	Renommée	Don.AllocNull
OPTIM-Productivité1	Suprimée	Trop générale
OPTIM-Productivité2	Suprimée	Trop générale
OPTIM-Set	Renommée	Don.Ensemble
OPTIM-SousExpr	Commonalisée	Qa.Factorisation
OPTIM-Strings	Renommée	QA.OptimStrings
OPTIM-Synchro	Renommée	QA.Synchro
ORGANI-Import	Renommée	Org.Importation
ORGANI-Package	Commonalisée	Org.DonneesOper
PORTAB-AppProperty	Renommée	QA.AppProperty
PORTAB-ConstPlat	Commonalisée	Int.Environement
PORTAB-DependAPI	Renommée	QA.Officiel
PORTAB-Deprecated	Renommée	QA.Deprecated
PORTAB-GetProperty	Renommée	QA.PortGetProperty
PORTAB-GUICapa	Commonalisée	Org.MatérielIndep

Identification Version 4	Nature	Nouvelle identification
PORTAB-GUIEvent	Renommée	QA.GUIEvent
PORTAB-GUIFonts	Renommée	QA.GUIFonts
PORTAB-GUIPaintGC	Renommée	QA.GUIPaintGC
PORTAB-GUISize	Renommée	QA.GUISize
PORTAB-IHMStable1	Renommée	QA.IHMStable1
PORTAB-IHMStable2	Renommée	QA.IHMStable2
PORTAB-InOutErr	Commonalisée	Org.MatérielIndep
PORTAB-JavaOfficiel	Renommée	QA.Officiel
PORTAB-Limit	Commonalisée	Org.MatérielIndep
PORTAB-Native	Renommée	QA.PortNative
PORTAB-Swing	Renommée	QA.Swing
PORTAB-SystemCall	Renommée	QA.SystemCall
SECUR-Applet	Renommée	QA.SecurApplet
SECUR-AppletTest	Renommée	QA.AppletTest
SECUR-Application	Renommée	QA.SecurApplication
SECUR-CLASSPATH	Renommée	QA.CLASSPATH
SECUR-JavaFlaws	Renommée	QA.JavaFlaws
SECUR-Obfuscator	Renommée	QA.Obfuscator
SPECS-Version	Renommée	QA.VersionNavig
STYLE-Blocs	Renommée	Tr.Blocs
STYLE-Langue	Commonalisée	Id.IdentRegle
STYLE-Tableau	Renommée	Don.TableauxDecl
THREAD-Création	Renommée	Dyn.ThreadCreation
THREAD-Daemon	Renommée	Dyn.Daemon
THREAD-Encapsuler	Commonalisée	Dyn.SectionCrtique
THREAD-SystemExit	Renommée	Dyn.Exit
VARLOC-Initialisation	Commonalisée	Don.Initialisation
VARLOC-Ligne	Commonalisée	Don.Separee
VARLOC-Proximité	Commonalisée	Don.Localite
VARLOC-Utilisation	Commonalisée	Don.LocalUnique

REFERENTIEL NORMATIF REALISE PAR :

Centre National d'Etudes Spatiales

Inspection Générale Direction de la Fonction Qualité

18 Avenue Edouard Belin

31401 TOULOUSE CEDEX 9

Tél. : 61 27 31 31 - Fax : 61 28 28 49

CENTRE NATIONAL D'ETUDES SPATIALES

Siège social : 2 pl. Maurice Quentin 75039 Paris cedex 01 / Tel. (33) 01 44 76 75 00 / Fax : 01 44 46 76 76
RCS Paris B 775 665 912 / Siret : 775 665 912 00082 / Code APE 731Z