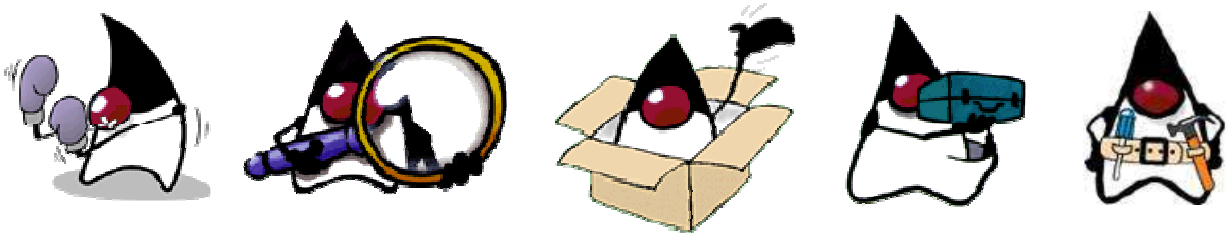
Laboratoire de Java (I)
[cours : Réseaux et technologie Internet]
Projet "InPrES Bus"

2ème informatique de gestion et
2ème informatique et systèmes (industrielle et réseaux)
2010-2011

Claude Vilvens – Christophe Charlet – Mounawar Madani –
Denys Mercenier – Jean-Marc Wagner



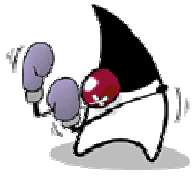
A. Introduction et contexte de développement

Le contexte général est celui d'une gare de bus : le projet considéré (nommé tout simplement "Inpres Bus") propose deux applications :

- l'application "**GestionInpresBus**" : une aide à la gestion des horaires des bus d'un groupe local de transport en commun;
- l'application "**VoyageurInpresBus**" : un outil de visualisation "en temps réel" des bus au départ de la gare centrale des bus, avec notification des incidents qui peuvent survenir.et occasionner des retards.

Seront donc essentiellement abordés ici les bases de Java, les interfaces graphiques AWT et surtout Swing, les classes utilitaires, les flux, les Java Beans, la portabilité Windows-Unix et une première approche des threads.

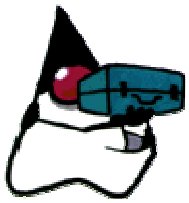
L'ensemble des travaux proposés ici est à réaliser en utilisant l'environnement de développement **NetBeans version 6.***, basé sur le **JDK 1.6**). Comme éditeur de texte plat, on pourra utiliser **JEdit 4.2** et **Xming 6.9** sera le serveur X-Window privilégié. Tous sont disponibles sur le serveur de ressources InPrES 10.59.5.223 (login="bac2", password="22xx", répertoire Java et sous-répertoires associés), de même que les pages d'aide HTML de la librairie Java (javadocs). Le dossier est, *à priori*, à réaliser **par équipe de deux étudiants** mais **peut aussi être présenté par un étudiant seul** qui le désirerait.



Un petit conseil : ***lisez bien l'ensemble de l'énoncé*** avant de concevoir (d'abord) ou de programmer (après) une seule ligne ;-). Ceci vous permettra non seulement d'avoir une vision globale du projet mais aussi de déjà remarquer des traitements communs à des points différents des applications. Prévoyez une schématisation des diverses classes (peut-être des diagrammes de classes UML ?) et élaborez d'abord "sur papier" (donc sans programmer directement) les divers scénarios correspondant aux fonctionnalités demandées.

B. Fonctionnalités attendues pour le projet "*InPrES Bus*"

Les applications demandées doivent fonctionner sur une machine **Windows (PC ou portable) et** sur une machine **UNIX (Sunray)** de l'InPrES.



Après discussion avec les futurs utilisateurs de l'application, une analyse sommaire a conduit à dégager que l'interface de l'application pourrait comporter les items de menu suivants :

Bus	**Conducteurs**	**Lignes**	**Horaires**	**Aide**
Nouveau	Nouveau	Définir	Ajouter une ligne horaire	Prise de notes
En/Hors service	Affectations	Modifier arrêt*	_____	Afficher le log
_____	_____	_____	Lister les lignes horaires	_____
Liste	Liste	Liste	Lister les incidents	A propos

Visuellement, elle a un aspect du type suivant (ce qui sera affiché dans la fenêtre principale est laissé à l'appréciation des développeurs) :

Chaque item de menu principal se voit associer une image caractéristique que l'on retrouve dans les items de sous-menu associés. On s'en doute, chaque sélection d'un item de menu fera apparaître une boîte de dialogue en rapport avec la fonctionnalité associée.

A chaque lancement de l'application, l'utilisateur se voit tout d'abord proposer une boîte de dialogue demandant à l'utilisateur de s'authentifier (sous la forme classique login-password) :

C. Par où commencer ? Séquence, précisions et conseils pour le développement

1ère évaluation : les bases du développement et du déploiement en Java

Il s'agit ci

♦ de maîtriser les **techniques fondamentales** de compilation, de déploiement, d'exécution et de traçage d'applications Java (javac, java, jar);
♦ de s'initier au mécanisme des événements au travers de la librairie basique d'interfaces graphiques **AWT**.

1. Histoires de jars

Pour ce petit exercice, on ne se servira pas de code généré et d'outil intégré de compilation-exécution proposé par Netbeans (ou d'un autre EDI comme Eclipse) : tout se fera par simple édition (avec un éditeur simple comme JEdit – disponible sur le serveur de ressources de l'InPrES 10.59.5.223), une fenêtre DOS (pour Windows) et une fenêtre (d'émulation Putty) de terminal (pour Unix).

L'objectif est ici clairement et uniquement de maîtriser les concepts de base de conception d'un GUI élémentaire et de déploiement (en termes de répertoires, jars, CLASSPATH, etc) d'une application utilisant une librairie, permettant ainsi dans la suite du laboratoire une utilisation raisonnée de Netbeans.

Le petit travail demandé sera **évalué au cours de la deuxième séance de laboratoire** (autrement dit, il y aurait intérêt à préparer avant ;-)). Voici la séquence des opérations :

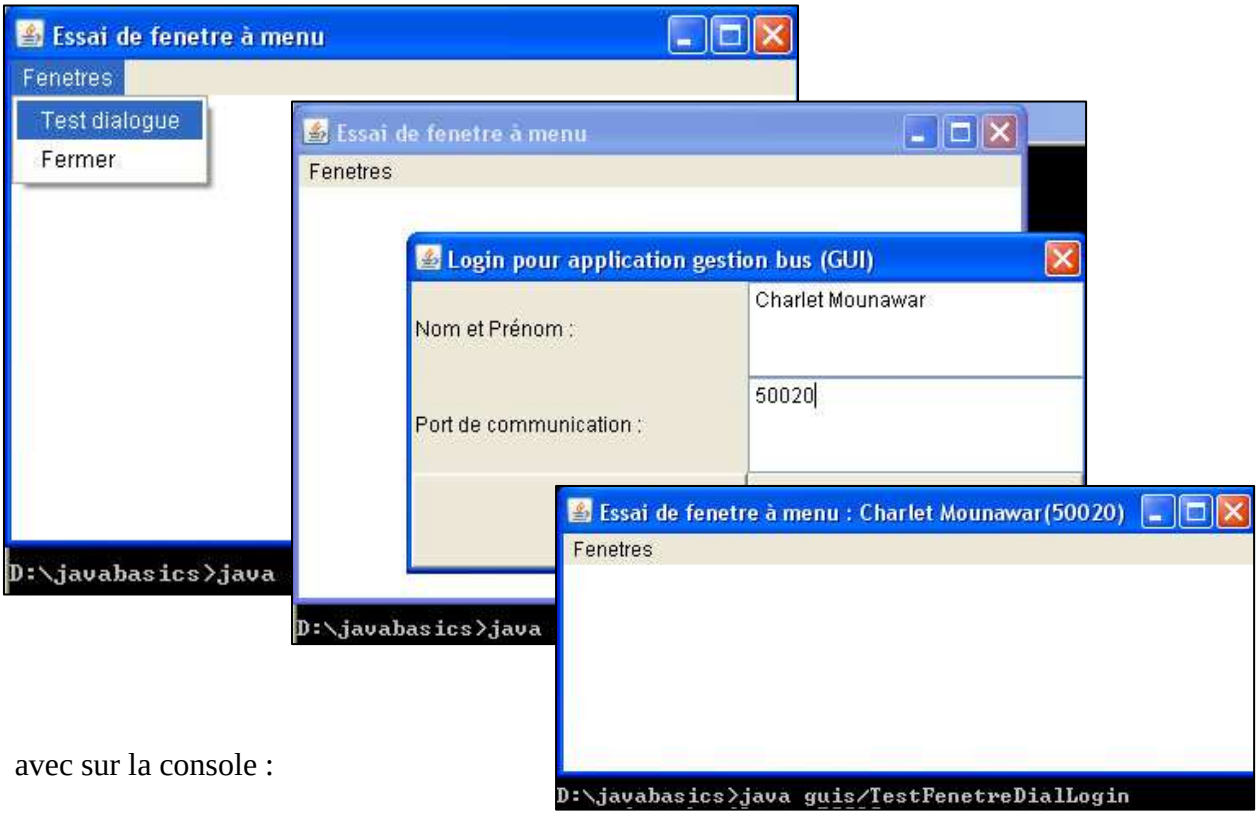
1) Dans un répertoire Javabasics, créer une classe **DialLogin**, GUI de type boîte de dialogue, appartenant au package **guis** et utilisant des composants AWT. Elle permet à un utilisateur de se faire reconnaître par l'application sur base de ses nom et prénom (une seule chaîne séparée par un espace) et d'un mot de passe (comme sur l'Ecole virtuelle). Il est invité aussi à introduire un numéro de port réseau (en fait, un simple nombre entier positif sous forme de chaîne de caractères – pour quoi faire ? voir plus bas). On veillera à utiliser le minimum de variables membres. Deux boutons complètent le GUI (donc structuré en un GridLayout de 4 lignes x 2 colonnes) : Ok (on peut alors récupérer le nom-prénom et le numéro de port choisis) et Annuler. Dans les deux cas, la boîte de dialogue est refermée …

Compiler cette classe en ligne de commande dans une fenêtre DOS. Vérifier au préalable en quelle version du JDK de Java on travaille : ce doit être 1.6_XX.

2) Une classe **TestConsoleDialLogin** (même package) comporte la méthode main() et ne sert qu'à tester la boîte DialLogin : elle fait apparaître la boîte et récupère les chaînes entrées pour les afficher dans la console. Compiler cette classe en ligne de commande dans une fenêtre DOS. Lancer l'exécution de l'application formée de ces deux classes - le résultat devrait ressembler à ceci :


```
D:\javabasics>java guis/TestConsoleDialLogin
User =
Choix du port = 50010
```

3) De même, une classe **TestFenetreDialLogin** (même package) comporte la méthode main() et ne sert qu'à tester la boîte DialLogin : elle fait apparaître la boîte et récupère les chaînes entrées pour ajouter les nom et prénom dans le titre et le port dans une zone d'affichage de la fenêtre. Compiler cette classe en ligne de commande dans une fenêtre DOS. Lancer l'exécution de l'application formée de ces deux classes - le résultat devrait ressembler à ceci :



avec sur la console :

```
D:\javabasics>java guis/TestConsoleDialLogin
```

User = Charlet Mounawar
Choix du port = 50010

3) Construire un jar **TestFenetreDialLogin.jar** exécutable avec cette application et l'exécuter. Le manifeste du jar devrait au moins contenir deux lignes du type suivant :

```
Manifest-Version: 1.0
Created-By: 1.6.0_13 (Sun Microsystems Inc.)
```

4) Porter le jar sous Unix (machine Sunray) et l'exécuter

- sur un terminal Sun;
- en émulation de terminal avec XMing comme serveur X-Window.

5) Construire un deuxième package **utils** qui ne contiendra pour l'instant que la classe **StringChecker** dont le rôle est de permettre de vérifier, au moyen de ses diverses méthodes, qu'une chaîne de caractères satisfait à telle ou telle condition – dans notre cas, il nous suffira de vérifier que la chaine précisée dans le constructeur ne comporte que des chiffres et représente un nombre qui est numéro de port valide (donc entier positif <65536). Une fois cette classe développée, il faut l'empaqueter dans un jar **LoginUtils.jar**.

6) La classe **ValidPortCheckerTest** (du même package) à réaliser ensuite ne sert évidemment qu'à tester la classe StringChecker utilisée sous forme de jar.

7) Dernière étape : la classe DialLogin, au moment où l'on appuie sur le bouton Ok, traite le port entré avec la méthode de vérification de StringChecker contenue dans LoginUtils.jar et affiche la conclusion sur la console. Une fois cette modification apportée, on crée un **TestFenetreDialLogin2.jar** pour qu'il puisse prendre en compte l'autre jar. Il n'y a plus qu'à vérifier que cela fonctionne … et à porter tout cela sous Unix.

Ouf ! …

2ème évaluation : les bases et les premières fonctionnalités de l'application

Il s'agit ici

♦ de maîtriser parfaitement la **conception orientée objets** d'un groupe de classes et interfaces modélisant des données utilisées par l'application;
♦ de se familiariser avec les techniques courantes d'utilisation des GUIs développés avec la **librairie Swing**.

2. Les bus, les horaires et leur contexte : les classes et les interfaces de base

Indépendamment des interfaces graphiques, on aura besoin des interfaces et classes suivants, tous placés dans le *package* **data**.

2.1) La représentation d'un bus est évidemment la première chose à définir. En fait, la classe **Bus**

♦ hérite de la *classe* **VehiculeTransportPassagers** (tout comme les classes Voiture, Bateau ou Wagon); une fonctionnalité classique est de savoir fournir le nombre maximum de passagers transportés;
♦ cette classe hérite de la *classe abstraite* **VehiculeTransport**, tout comme la classe VehiculeTransportMarchandises ou VehiculeTransportMixte; une fonctionnalité classique est de savoir préciser si le véhicule est en panne ou pas;
♦ cette classe VehiculeTransport implémente l'*interface* **ObjetMateriel** qui déclare la méthode

Dimensions getDimensions()

où **Dimensions** est un *interface* déclarant la méthode

String getUnite()

et dont une implémentation est la *classe* **Dimensions3D** aux prévisibles méthodes getLongeur(), getLargeur() et getHauteur().

En soi-même, les renseignements concernant un bus peuvent être très nombreux mais nous nous limiterons aux suivants :

♦ numéro d'identification (du bus, pas de sa ligne ! – à la réflexion, un tel numéro a un sens pour tout VehiculeTransport …);
♦ le fait d'être articulé ou pas;
♦ le fait d'être en service ou pas
♦ le nom du dépôt.

2.2) Passons à présent au conducteur du bus. Bien sûr, il a un nom et un matricule. Mais ces notions sont partageables avec d'autres classes, si bien que l'on suivra la hiérarchie suivante :

♦ l'*interface* **EstIdentifiable** déclare la méthode

String getIdentifiant()

♦ la *classe* **Personne** implémente cet interface (à remarquer qu'un VehiculeTransport aussi !) et ajoute le nom, le prénom et la date de naissance;

♦ la *classe* **Conducteur** (qui hérite logiquement de Personne) retourne comme identifiant le numéro de matricule du conducteur et est capable de fournir, le cas échéant, le bus que le conducteur conduit habituellement.

2.3) La *classe* **Ligne** matérialise une ligne de bus. Bien logiquement, elle comporte un numéro, un nom (du genre : "Seraing" pour la ligne 2, "Fléron" pour la ligne 10) et une liste ordonnée d'objets **Segment** qui définit le parcours suivi par cette ligne. Un Segment est donc constitué de deux arrêts successifs (par exemple, "place Kuborn" et "rue Ferrer – début") et de la durée du trajet (en minutes).

2.4) Un **Horaire** est bien sûr constitué d'un certain nombre d'objets **LigneHoraire**, objet qui comporte

- ♦ un bus;
- ♦ un conducteur;
- ♦ une ligne;
- ♦ une date-heure de départ.

Toutes ces classes seront testées par des instanciations au moyen de constantes (pas d'entrée clavier). Les affichages se feront sur la console.

Un sous-package **data.exceptions** contiendra les classes d'exception utilisées dans les différentes classes, par exemple :

- ♦ **MalFormedIdentifierException** : un identifiant qui contient autre chose que des lettres, des chiffres et -;
- ♦ **IncoherentDimensionsException** : une largeur qui est plus grande qu'une longueur;
- ♦ **EtcException**.

3. Les interfaces graphiques : la création de lignes

3.1 La définition d'une ligne

Nous allons tout d'abord nous intéresser à la définition d'une ligne (dans le menu : **3. Lignes** → Définir) Cette opération se fera avec un interface graphique du type suivant (ce n'est qu'une évocation – on peut adapter en fonction des besoins, des idées) :

Quelques précisions :

- on commence par définir les arrêts de la ligne (zone d'entrée "Nouvel arrêt" et boîte combo de visualisation et de suppression éventuelle); ils sont mémorisés en mémoire dans une liste liées (LinkedList);
- au cours de la définition des arrêts, on peut les visualiser dans une table (JTable);
- une fois la succession des arrêts confirmée définitivement, on peut attribuer une durée à chaque segment;
- tout au long de ces opérations, la ligne est construite visuellement dans un composant arbre (JTree);
- la ligne est définitivement mémorisée dans un objet instance de la classe Ligne (évoquée ci-dessus en 2.3) (on peut l'afficher en console pour vérifier la bonne construction de l'objet); ceci a seulement pour effet de mémoriser la ligne dans un container mémoire – dans la version 2, il sera sauvé dans une structure persistante comme un fichier.

3.2 A propos

Pour en terminer avec ce module, on pourra déjà implémenter la boîte de dialogue correspondant à

Aide→A propos

On y verra apparaître, outre le nom du (des) auteur(s), les dates de création et de dernière modification.

3.3 Portabilité

Pour rappel, afin d'illustrer la portabilité de Java,

l'application demandée doit fonctionner sur une machine **Windows** **et** sur une machine **UNIX** de l'INPRES.

On se servira de la console comme outil de trace/log en affichant les coordonnées des objets créés au moyen du GUI.

3ème évaluation : les fonctionnalités évoluées de l'application

Il s'agit ici

♦ de maîtriser les **techniques classiques de développement Java** : containers associatifs, flux, fichiers properties, sérialisation et cohérence des données;
♦ d'être capable de développer une chaîne d'événements dans le contexte des **Java Beans**;
♦ de savoir utiliser une **librairie extérieure** fournie dans un jar avec sa documentation en Javadoc;
♦ de se familiariser avec l'utilisation basique des **threads** en Java.

On se souviendra que projet "Inpres Bus" comporte deux applications : "GestionInpresBus" pour les horaires et "VoyageurInpresBus" pour informer les voyageurs de la gare de bus des départs et des incidents qui peuvent survenir et occasionner des retards.

4. L'application GestionInpresBus : confection des horaires

Il s'agit à présent de compléter l'application développée jusqu'ici ;-)

4.1 Les utilisateurs

A priori, il faudrait prévoir une gestion convenable des utilisateurs de l'application. Mais nous nous contenterons ici d'un mécanisme très simple (simpliste même ;-)) : les utilisateurs autorisés à se servir de l'application seront mémorisés dans un fichier **user.properties** structuré selon

<nom et prénom utilisateur> : <mot de passe>

4.2 Les lignes : la suite

a) Il s'agit tout d'abord d'ajouter aux fonctionnalités déjà existantes pour l'item

Lignes → Définir

la persistance des données qui leur faisait défaut. Il faudra donc simplement sérialiser la ligne créée par le mécanisme des objets persistants (Serializable). Un simple bouton "Enregistrer" provoquera l'enregistrement de la ligne dans un fichier portant un nom en rapport avec la ligne (par exemple : "ligne10" ou "ligne30B"). Un fichier texte **liste_fichiers_lignes.txt** additionnel contiendra la liste des noms de fichiers disponibles.

b) L'item de menu

Lignes → Liste

♦ permet évidement d'obtenir dans une boîte de dialogue la liste des lignes (par lecture du fichier liste_fichiers_lignes.txt). Un double clic sur une ligne fait apparaître dans une deuxième boîte de dialogue le détail sous forme arborescente (comme lors de la définition de la ligne). Pour gérer un double clic sur une liste, il suffit que l'objet JList traite l'événement MouseEvent selon :

```
public void mouseClicked (MouseEvent e)
{
      if (e.getClickCount() == 2)
      {
         int index = listeGeree.locationToIndex(e.getPoint());
         System.out.println("On a cliqué sur l'item n° " + index);
       }
}
```

4.3 Les bus

La gestion des bus sera limitée ici au strict nécessaire. La persistance des bus sera assurée par la sérialisation d'une Hashtable dont la clé sera le numéro de bus et la valeur l'objet Bus correspondant.

a) Bus → Nouveau permet évidemment de créer un nouveau bus et de l'ajouter à la hashtable.

b) Bus → En/Hors service : ne réclame sans doute pas de commentaire …

c) Bus → Liste : similaire à l'affichage des lignes.

4.4 Les conducteurs

La gestion des conducteurs sera elle aussi limitée au strict nécessaire et tout à fait similaire à celle des bus. Leur persistance sera donc assurée par la sérialisation d'une Hashtable dont la clé sera le nom-prénom du conducteur et la valeur l'objet Conducteur correspondant.

a) Conducteur → Nouveau permet évidemment de créer un nouveau Conducteur et de l'ajouter à la hashtable.

b) Conducteur → Affectation : permet d'associer un conducteur au bus qu'il conduit habituellement. Le conducteur et le bus sont choisis d'après les hashtable des conducteurs et des bus.

c) Conducteur → Liste : similaire à l'affichage des bus.

4.4 Les horaires : construire une ligne horaire

a) Une fois les trois premiers menus fonctionnels, il est assez simple d'implémenter

Horaires → Ajouter une ligne horaire

car, pour rappel, un objet LigneHoraire comporte

- un bus (en service !);
- un conducteur;
- une ligne;
- une date-heure de départ.

Les deux premiers éléments sont choisis dans des boîtes de liste, le numéro de ligne dans une boîte combo et la date heure d'une manière quelconque mais ne permettant pas d'entrer des informations vides de sens …

Les informations ainsi définies se répercutent dans l'interface de la fenêtre principale. L'horaire est simplement sérialisé à chaque étape, afin de pouvoir être récupéré au redémarrage de l'application.

b) L'item

Horaires → Lister les lignes horaires

consiste évidemment simplement à remplir une JTable avec les informations provenant de l'horaire sérialisé.

5. Communication entre GestionInpresBus et VoyageurInpresBus

5.1 Une application de visualisation minimaliste

C'est le moment de mettre en scène l'application VoyageurInpresBus qui est donc une application indépendante dont le but est d'afficher les informations sur les bus en partance. Son interface graphique est donc des plus simples :

- l'affichage des lignes horaires s'effectuant dans une liste (pour des raisons qui apparaîtront très bientôt).

Mais comment GestionInpresBus va-t-elle transmettre à VoyageurInpresBus la ligne horaire qu'elle a créé ?

5.2 La communication entre les deux applications

En fait, GestionInpresBus va envoyer à VoyageurInpresBus une chaîne de caractères (formée de la concaténation des informations de la ligne horaire) en utilisant une communication réseau TCP/IP (voilà pourquoi un numéro de port est demandé au login !).

Pour cela, il est fourni deux classes (appartenant au package network) au sein du fichier **BusInpresLib.jar** (disponible sur le serveur ftp cité au début de cet énoncé) :

♦ NetworkStringSender : classe permettant l'émission d'une chaîne de caractères vers un récepteur d'adresse IP et de port donnés.
♦ NetworkStringReceiver : classe utilisant un thread permettant la réception d'une chaîne de caractères qui sera affichée dans une JList.

Ces deux classes sont documentées par les javadocs qui se trouvent dans BusInpresLib.zip (disponible au même endroit).

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

network

Class NetworkStringReceiver

```
java.lang.Object
  └network.NetworkStringReceiver
```

```
public class NetworkStringReceiver
extends java.lang.Object
```

Classe utilisant un thread permettant la réception d'une chaîne de caractères qui sera affichée dans une JList.

Version:
1.0.
Author:
Claude Vilvens.

Constructor Summary

`NetworkStringReceiver(int p, javax.swing.JList l)`
Constructeur d'initialisation du récepteur.

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS — FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

network

Class NetworkStringSender

```
java.lang.Object
  └network.NetworkStringSender
```

```
public class NetworkStringSender
extends java.lang.Object
```

Classe permettant l'émission d'une chaîne de caractères vers un récepteur d'adresse IP et de port donnés.

Version:
1.0.
Author:
Claude Vilvens.

Constructor Summary

`NetworkStringSender(java.lang.String a, int p)`
Constructeur d'initialisation de l'envoyeur.

Method Summary	
java.lang.String	**getAdresse**() Accesseur read de l'adresse.
int	**getPort**() Accesseur read du port.
void	**sendString**(java.lang.String s) Méthode d'envoi.
void	**setAdresse**(java.lang.String adresse) Accesseur write de l'adresse.
void	**setPort**(int port) Accesseur write du port.

Leur utilisation dans le contexte de notre projet est très simple :

♦ VoyageurInpresBus instancie un NetworkStringReceiver à la fin de son constructeur (le port est donc entré par l'utilisateur lors du démarrage de l'application);
♦ GestionInpresBus possède une variable membre NetworkStringSender instanciée à la fin de son constructeur et utilise la méthode sendString() à la fin du traitement du menu

Horaires → Ajouter une ligne horaire

6. Les incidents par Java Beans

6.1 Des perturbations sur les lignes

Dans cette dernière partie, des incidents de nature à retarder les départs des bus (conditions climatiques comme neige ou verglas, grève imprévue, accident sur un parcours, panne, …) vont être simulés au moyen de Java Beans instanciés dans GestionInpresBus.

6.2 Le bean BusAlertGeneratorBean

Ce bean est à la base de la simulation. Une instance de BusAlertGeneratorBean tire des nombres aléatoires au hasard et vérifie s'ils sont multiples d'un nombre de référence passé à son constructeur. Si c'est le cas, il va tirer un deuxième nombre aléatoire entre 1 et NOMBRE_TYPE_INCIDENT (=5), chaque valeur correspondant à

constante	type d'incident
INCIDENT_NEIGE	la neige bloque la gare ou la destination
INCIDENT_VERGLAS	le verglas bloque la gare ou la destination
INCIDENT_GREVE	une grève sauvage bloque la gare
INCIDENT_ACCIDENT	un accident s'est produit sur la ligne : détournement à prévoir
INCIDENT_PANNE	un bus est en panne à la sortie de la gare : retard à prévoir

Toutes ces constantes sont définies dans l'interface BusConstants.

Le bean va ensuite générer un événement **BusAlertEvent** qui contiendra bien sûr la constante représentative de l'incident et un nombre entier représentant, le cas échéant, la ligne concernée.

Le listener de ce genre d'événement (**BusAlertListener**) ne possède que la méthode public void **NewBusAlertReceived** (MatchEvent e).

6.3 Le bean BusAlertAnalyzeBean

Ce second bean est un BusAlertListener : il s'agit d'une petite boîte de dialogue avertissant de l'incident et invitant le gestionnaire (l'utilisateur de l'application) à évaluer le retard occasionné (en minutes ou en heures) en fonction des informations qu'il aura pu se procurer par d'autres voies.

Si l'utilisateur valide l'incident, le bean va modifier sa propriété MessageHoraireATransmettre qui représente le message qui devra être transmis aux voyageurs et qui est une propriété liée (elle utilise donc un PropertyChangeSupport).

6.4 Le bean BusAlertSenderBean

Ce troisième bean est PropertyChangeListener pour la propriété liée du bean précédent : lorsqu'il reçoit le PropertyChangeEvent correspondant, il envoie le message à l'application VoyageurInpresBus en utilisant le NetworkStringSender de l'application. Il met aussi à jour l'interface graphique de l'application.

7. L'usage élémentaire des threads

Cependant, si l'on souhaite que la simulation soit convaincante, il faut que la génération d'un incident s'effectue en parallèle de l'utilisation de l'application : autrement dit, il nous faudrait un **thread** … notion déjà connue sous Unix et qui sera étudiée de manière exhaustive pour ce qui concerne Java en 3ème. Nous allons donc nous contenter du minimum en ce qui concerne ces threads Java.

7.1 Une classe Thread fournie

Il vous est fourni une classe **ThreadRandomGenerator** qui produit un nombre aléatoire dans la fonction qu'il exécute (en fait, sa méthode **run**() héritée de la classe **Thread** de la librairie Java) et "prévient" l'objet qui lui a été passé comme paramètre dans son constructeur, c'est-à-dire qu'il appelle une méthode de cet objet nommée *changeNombre*(). Bien sûr, pour assurer que l'objet en question possède bien cette méthode, il devra implémenter un interface **UtilisateurNombre** qui ne déclare que deux méthodes :

UtilisateurNombre.java

```
package Threadsutils;

public interface UtilisateurNombre
{
   String getIdentifiant();
   void changeNombre(int n);
}
```

Donc, en clair, AlertGeneratorBean implémente cet interface. Le constructeur du thread :

public **ThreadRandomGenerator**(UtilisateurNombre un, int bi, int bs, int md, int tp)

réclame, outre l'objet à "prévenir", les bornes inférieures et supérieures des nombres aléatoires produits, le nombre multiple de déclenchement et le temps de pause entre deux générations. En effet, pour que la simulation soit assez aisée à manipuler, on "freine" le déroulement des opérations de génération au moyen de l'appel de la méthode de classe **sleep**(<nombre de millisecondes>);

(qui réclame un traitement d'exception).

En résumé :

ThreadRandomGenerator.java

```
package Threadsutils;;

/**
 * @author  Vilvens
 */

public class ThreadRandomGenerator extends java.lang.Thread
{
   private UtilisateurNombre utilisateurThread;
   private int borneInferieure, borneSuperieure, multipleDeclenchement, tempsPause;
   private int nombreProduit;

   public ThreadRandomGenerator(UtilisateurNombre un, int bi, int bs, int md, int tp)
   {
      utilisateurThread = un; borneInferieure=bi; borneSuperieure=bs;
      multipleDeclenchement = md; tempsPause = tp; nombreProduit = -1;
   }

   public void run()
   {
         Double dr;
         while (true)
         {
```

```
                dr = new Double(borneInferieure +
                        Math.random()*(borneSuperieure - borneInferieure));
                nombreProduit = dr.intValue();
                System.out.println(utilisateurThread.getIdentifiant() +
                        "> nombreProduit = " + nombreProduit);


                if (nombreProduit % multipleDeclenchement == 0)
                {
                   System.out.println(utilisateurThread.getIdentifiant() +
                        "> ---------- !!!!!!! " + nombreProduit + "!!!!");
                   utilisateurThread.changeNombre(nombreProduit);
                }

            try
            {
               Thread.sleep(tempsPause*1000);
            }
            catch (InterruptedException e)
            {
               System.out.println("Erreur de thread interrompu : " + e.getMessage());
            }
      }
   }
}
```

AlertGeneratorBean implémente **UtilisateurNombre** et, dans son constructeur, instancie un tel thread ThreadRandomGenerator (pour réaliser le travail de génération et de test de multiple de déclenchement) puis le fait démarrer par l'appel de la méthode **start**() de celui-ci (héritée de la classe **Thread** de la librairie Java).

Remarque
On pourrait craindre que l'envoi de messages d'incident à l'application VoyageurInpresBus ne perturbe l'envoi de lignes horaires : dans ce cas, un mécanisme de synchronisation devrait bien sûr être mis en place (méthodes synchronized : voir cours de 3ème année). Mais, en fait, si la boîte de dialogue de validation est modale, l'utilisateur ne pourra pas envoyer simultanément les deux types de messages et le problème ne se pose donc pas vraiment.

8. Quelques spécifications supplémentaires

1) Idéalement, l'application dépend d'un fichier de configuration qui est également un autre ***fichier properties*** : on y trouve des indications telles que

- le nom des fichiers properties et de sérialisations;
- le fait de savoir si les dates sont à afficher en format francophone ou anglophone, abrégé ou long;
- le nombre de référence utilisé pour les simulations par tirage aléatoire;
- le nom du fichier log.

2) A priori, les développements proposés ci-dessus s'effectuent avec NetBeans 6.* sous Windows. Une fois réalisée, on vérifiera que l'on sait exécuter l'application depuis une fenêtre DOS (technique des class en jar et java –jar). On demande ensuite de porter l'application sous UNIX et de la tester sur la machine Sunray.

3) En fin de réalisation, on générera les Javadocs associés à l'application.

9. Les items du menu d'aide

La signification de la plupart des items de menus est évidente. Précisons cependant que :

- Aide → Prise de notes : il s'agit de pouvoir créer un fichier texte contenant quelques notes;
- Aide → Afficher le log : les traces sur la console sont remplacées par un fichier de log; l'utilitaire de log est un JavaBean **FichierLog**; ce bean ne génère pas d'événement et est donc simplement une classe qui écrit et lit dans un fichier texte (Reader et Writer); on utilisera bien sûr un JTextArea pour la visualisation;

D. Principaux points de l'évaluation

1. Développements et déploiements en ligne de commande (Eval 1)	
classes **DialLogin**, **TestFenetreLogin et TestConsoleDialLogin**	
compilation et exécution	
TestFenetreDialLogin.jar : création et utilisation sous DOS; java -jar	
portage du jar sous Unix; java –jar	
classes **StringChecker ValidPortCheckerTest** (dans **LoginUtils**.jar)	
TestFenetreDialLogin2.jar utilisant **LoginUtils**.jar	
2. Les classes et les interfaces (Eval 2)	
Classe Bus Classe VehiculeTransportPassagers + méthode nbr de passagers max. VehiculeTransportMarchandises ou VehiculeTransportMixte Classe abstraite VehiculeTransport + fct en panne ou pas ? Classe ObjetMateriel Interface Dimensions Classe Dimensions3D	
Classe personne Interface EstIdentifiable Classe conducteur	
Classe Ligne Classe Horaire Classe Segment Classe HoraireLigne	
MalFormedIdentifierException IncoherentDimensionsException EtcException.	
3. La fenêtre de l'application et les premiers traitements (Eval 2)	
fenêtre principale de base avec menus (items avec images)	
Boîte A propos	
phase de login (utilisateurs gérés en mémoire seulement – à revoir plus loin avec les fichiers nécessaires)	
utilisation sous **UNIX** – java –jar	
Interface graphique « Définition d'une ligne » Utilisation Jtree pour visualiser la ligne et Jtable LinkedList pour les arrêts	
4. Fenêtre de l'application complétée (Eval 3) **Les utilisateurs (Eval 3)**	
Sérialisation dans un fichier properties	
Items principaux de l'application (Eval 3)	
Lignes Liste + événement MouseClicked pour afficher sous forme arborescente Sérialisation	
Bus Ajout/HS/Liste + sérialisation Hashtable	
Conducteurs Ajout/affectation/liste + sérialisation Hashtable	
5. La communication vers VoyageurInpresBus (Eval 3)	
Utilisation BusInpresLib.jar et envoie à VoyageurInpresBus	

6. Les beans (Eval 3)	
BusAlertGeneratorBean (+ classe **BusAlertEvent**)	
BusAlertAnalyzeBean (+ interface **BusAlertListener**)	
BusAlertSenderBean (+ propriété liée)	
7.Utilisation de ThreadRandomGenerator	
8. Divers (Eval 3)	
Utilisation de l'application sous **UNIX** : java –jar et utilisation des properties du système (file.separator, user.home, etc)	
fichier log	
Paramétrisation de l'application via properties	
javadocs du projet	
utilisation des classes Date et DateFormat	
9. Les informations (Eval 3)	
JTextArea et Reader/writer pour log	
Prise de notes	
A propos	

E. Echéances

1) pour le point 1 : **début de la deuxième séance de laboratoire** (semaine du 28 mars 2011) avec un **document papier** expliquant les diverses opérations réalisées.

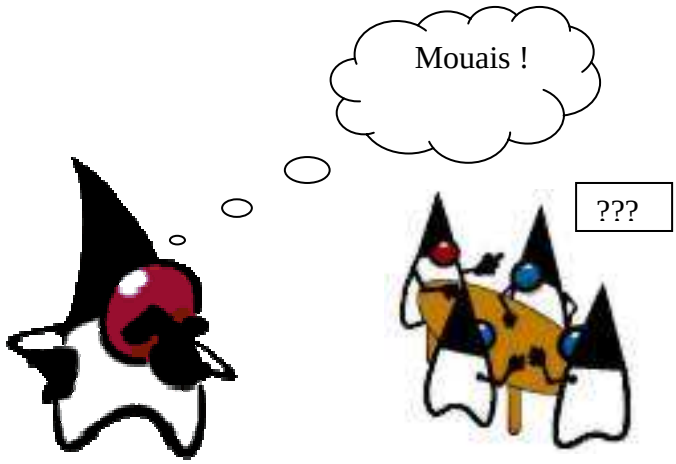
2) pour les points 2 et 3 :

Date de rentrée du dossier (papier) : **le lundi 25 avril** 2011 à 8h20 au plus tard (ce peut être plus tôt).
Test et évaluation de ce dossier : durant les laboratoires à partir de cette date **OU** sur rendez-vous avec le professeur de laboratoire
Remarque : tout dossier déposé dans les temps mais non évalué dans un délai de 2 semaines est sans valeur !

3) pour les points 4 à 10 :

Test et évaluation de ce dossier : examen de fin d'année selon horaire d'examen **OU** plus tôt pour les étudiants qui ont terminé (prendre rendez-vous avec le professeur de laboratoire).

Bon travail !



Et n'oubliez de rentrer votre contrat pédagogique au professeur de laboratoire *(ci-dessous)!!*

s: CV, CC, DM, MM & JMW.

Haute Ecole de la Province de Liège
Année académique 2010-2011
2ème bachelier en informatique de gestion, informatique et systèmes/informatique industrielle, informatique et systèmes/réseaux et télécommunications

cours : **Réseaux et technologies Internet**
enseignants : C.Vilvens, C.Charlet, M.Madani, D.Mercenier, JM.Wagner

Contrat pédagogique

1) L'évaluation établissant la note du cours de "Réseaux et technologie Internet" est réalisée de la manière suivante :

- théorie : un examen écrit en juin 2011, sur base des notes de cours ("Java I: programmation de base) et de questions exemples (liste non exhaustive) fournies en mai et à préparer; il sera coté sur 20;
- laboratoire : 3 évaluations (aux dates précisées ci-dessous), chacune cotée sur 20; la **moyenne pondérée de ces 3 cotes** (poids respectifs de **2/10**, **3/10** et **5/10**) fournit une note de laboratoire sur 20;
- note finale : **moyenne pondérée de la note de théorie (poids de 4/10) et de la note de laboratoire (poids de 6/10)**.

Cette procédure est d'application tant en 1ère qu'en 2ème session, ainsi que lors d'une éventuelle prolongation de session.

2) *Dans le cas où les travaux sont présentés par une équipe de deux étudiants*, chacun d'entre eux doit être capable d'expliquer et de justifier l'intégralité du travail (pas seulement les parties du travail sur lesquelles il aurait plus particulièrement travaillé).

3) En 2ème session et en prolongation de session, un **report de note** est possible pour chacune des trois notes de laboratoire ainsi que pour la note de théorie **pour des notes supérieures ou égales à 10/20**.
Toutes les évaluations (théorie ou laboratoire) ayant des **notes inférieures à 10/20** sont **à représenter dans leur intégralité**.

La première partie des travaux de programmation réseaux sera **évaluée** par l'un des professeurs du laboratoire **à partir du 28 mars 2011** (avec rentrée d'un dossier papier - le délai est à *respecter* impérativement).

La deuxième partie de ces travaux sera **évaluée** par l'un des professeurs du laboratoire **à partir du 25 avril 2011** (avec rentrée d'un dossier papier - le délai est toujours à *respecter* impérativement).

La troisième partie sera **évaluée** lors de l'examen de laboratoire en juin 2011 (le dossier papier n'est plus nécessaire).

Je soussigné :

Nom : …….. Prénom: ……

Section : …… bachelier en ……. Groupe : 22….

déclare avoir pris connaissance du contrat pédagogique concernant le cours de Réseaux et technologies Internet de 2ème année en bachelier en informatique (toutes options).

Date et signature :