Interface-utilisateur graphique d'une application Java

DUT Informatique 1ère année

Henri Garreta & Cyril Pain-Barre

Introduction

Ces travaux pratiques sont une initiation à la programmation des interfaces homme-machine graphiques (ou *GUI*, pour *Graphics User Interface*) en Java.

Nous étudierons ici *JFC/Swing*, la bibliothèque la plus répandue, développée par Sun Microsystems Inc., la maison mère de Java, et gratuitement distribuée par Oracle Corporation. Avant *Swing* il y a eu *AWT*, une bibliothèque moins perfectionnée, mais sur laquelle on ne peut pas faire l'impasse car Swing est bâtie sur *AWT* et de nombreux éléments de cette dernière se retrouvent tels quels dans *Swing*.

Outre quelques supports succincts mis à votre disposition, les principaux documents dont vous aurez besoin pour réaliser ces TP sont les deux suivants, très importants :

- la documentation en ligne de l'API Java, le « manuel de référence », tout à fait indispensable pour réaliser n'importe quel programme Java ; c'est une partie de l'importante documentation officielle de Java.
- le tutoriel Java, *The Java Tutorial*, et tout spécialement sa partie *Creating a GUI with JFC/Swing*, comportant de nombreuses indications et de précieux exemples à propos des composants graphiques.

AWT est formée des classes des paquetages dont les noms commencent par **`java.awt`** (**`java.awt`**, **`java.awt.event`**, etc.). *Swing* est constituée par les classes des paquetages dont les noms commencent par **`javax.swing`** (**`javax.swing`**, **`javax.swing.border`**, etc.). Tous ces paquetages et classes sont entièrement expliquées dans la documentation en ligne mentionnée plus haut.

Mode opératoire

L'objet de chacun des exercices suivants est la réalisation d'une application Java, qu'il faut concevoir, saisir, compiler et tester *exhaustivement*.

Pour saisir et exécuter (voire déboguer) vos programmes, nous recommandons vivement l'utilisation d'*Eclipse*, qui facilite grandement le travail du développeur. Dans cette optique, les documents suivants peuvent s'avérer utiles :

- Sur l'installation et utilisation d'eclipse
- Utilisation du débogueur d'eclipse

→ Si, comme c'est recommandé, vous développez vos programmes en utilisant *eclipse*, nous vous conseillons de définir des paquetages différents (nommés **`exo_01_01`**, **`exo_01_02`**, etc.). *Eclipse* se chargera pour vous de créer les répertoires correspondants et d'y placer les fichiers sources

→ Si vous n'utilisez pas *eclipse* (on se demande bien pourquoi), vous devrez créer *manuellement* des répertoires **`exo_01_01`**, **`exo_01_02`**, etc., pour y placer vos fichiers sources. Il sera alors plus simple pour vous **de ne pas** mettre en tête de chaque fichier une instruction **`package`** plaçant chaque classe dans un paquetage spécifique.

TP 1 - Pour Débuter

Objectif. Le but de ce TP est de faire ses premiers pas dans la programmation Java et, en particulier, de s'assurer qu'on a bien compris comment :

- saisir, compiler et exécuter une application Java,
- trouver l'information sur les classes et leurs méthodes en parcourant la documentation en ligne de l'API,
- utiliser certaines classes utilitaires d'intérêt général, comme les collections et les itérateurs,
- programmer la « partie obligée » (le cadre) de toute application ayant une interface graphique.

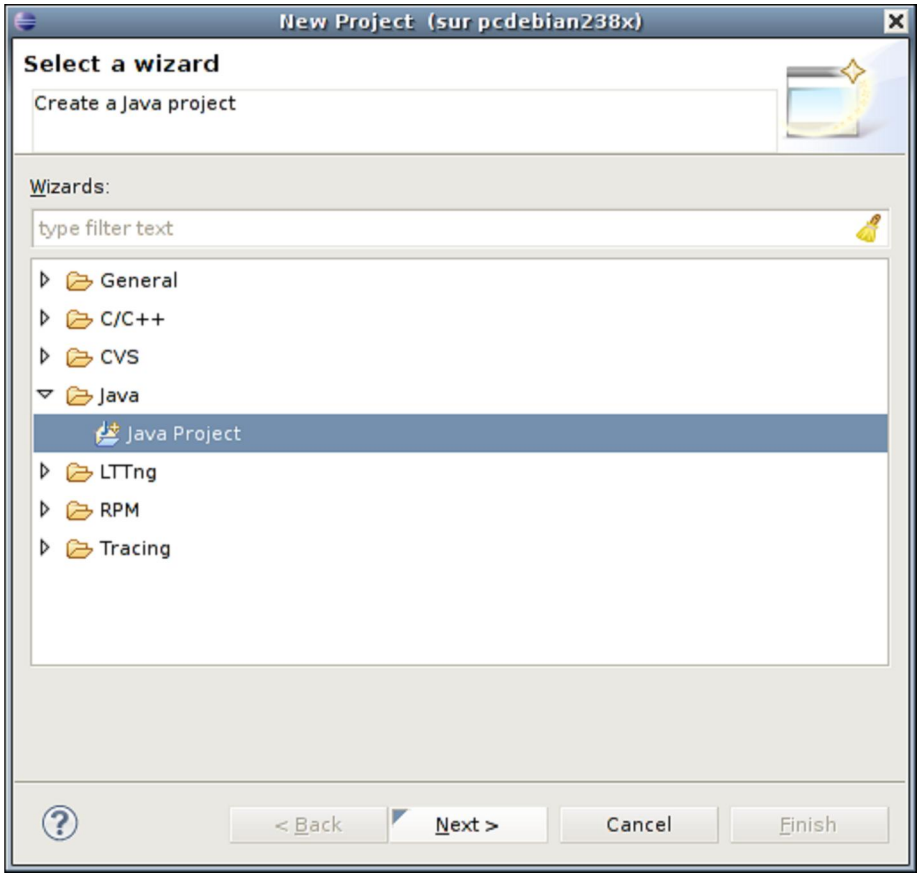
1.1 - Premières applications Java

Exo 1.1.1 - L'inévitable "Print Hello"

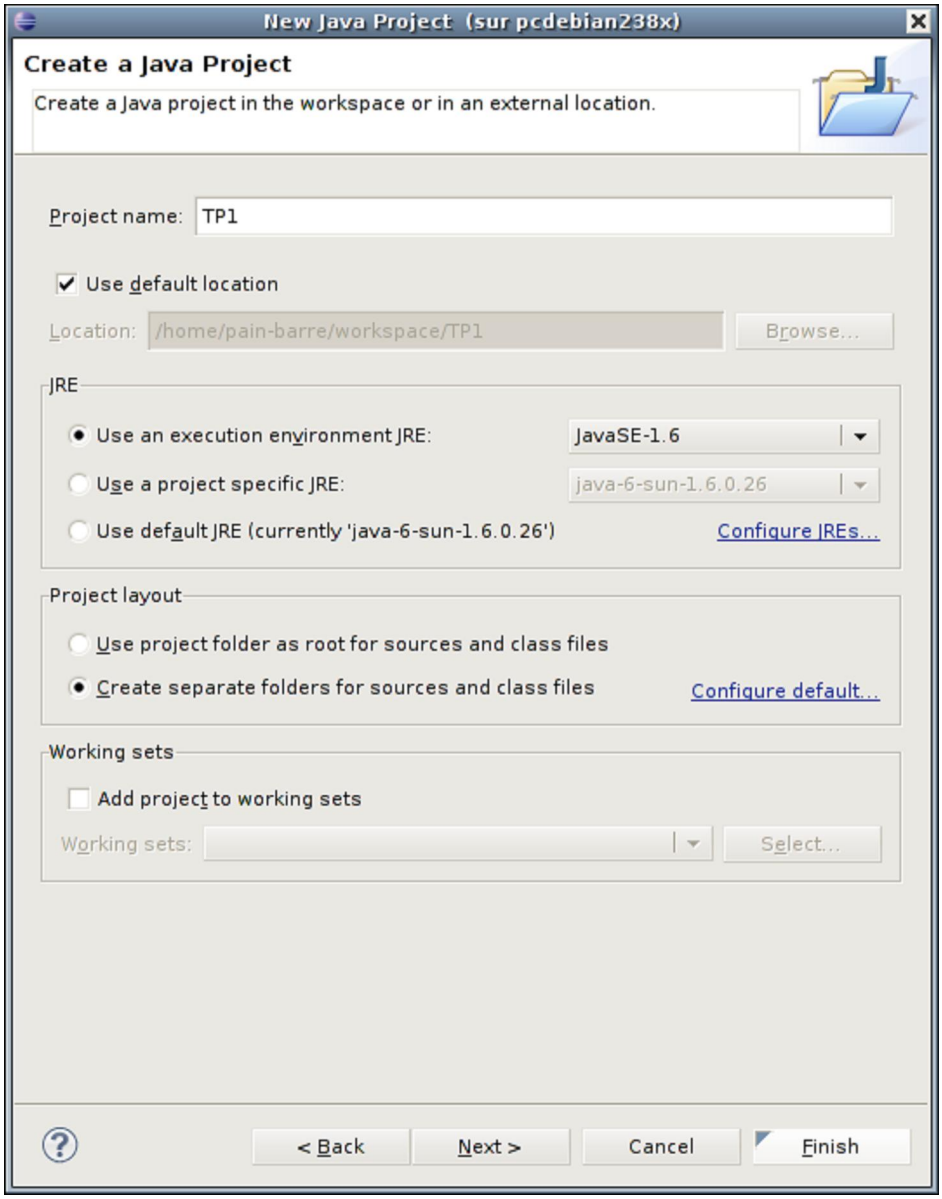
[Corrigé]

Nous allons créer notre première application Java. Pour cela :

- commencer par lancer *Eclipse* (version Java et non pas C/C++). Choisir/créer un *workspace* quelconque **en dehors de `net-home`** (il faudra penser à sauvegarder son travail en fin de séance). Sur la fenêtre affichée, cliquer sur l'icône représentant une flèche enroulée pour ouvrir le *Workbench* (environnement de développement)
- choisir la création d'un projet Java qui, selon la *perspective* (affichage) actuelle d'*eclipse*, se demande directement via le menu *File → New → Java Project*, ou un peu moins directement via le menu *File → New → Project* et en choisissant *Java → Java Project* :

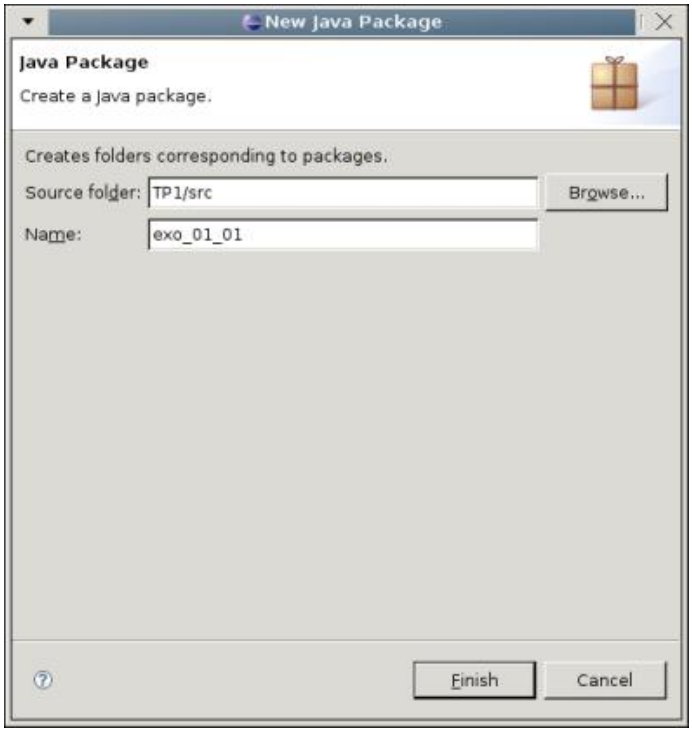


Sur la fenêtre de création du projet, saisir le nom **`TP1`**, s'assurer que le JRE est bien **`JavaSE-1.6`** et que le bouton radio demandant de séparer les sources et les binaires est bien sélectionné, puis cliquer sur *Finish* :



Le projet est alors présenté dans la perspective de développement Java.

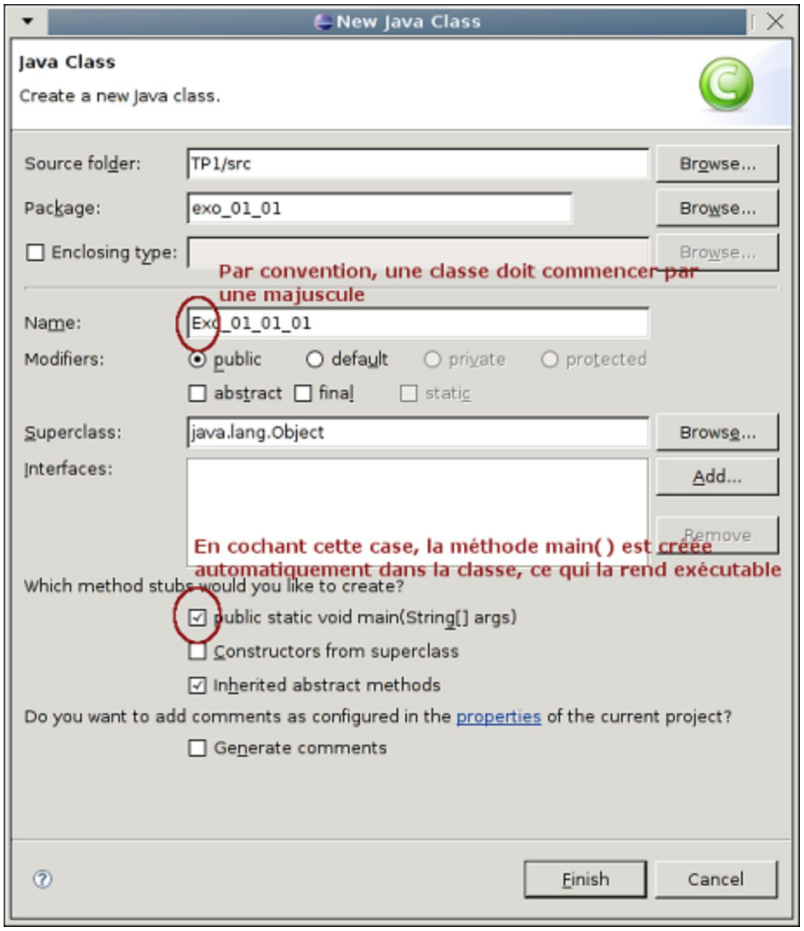
- dans la zone *Package Explorer* (sur la gauche), effectuer un clic droit sur votre projet et choisir *New* → *Package* pour créer un package (paquetage) **`exo_01_01`** :



- toujours dans la zone *Package Explorer*, effectuer un clic droit sur ce package et choisir *New* → *Class* pour créer une **classe exécutable** `Exo_01_01_01`. Une classe exécutable est une classe possédant une méthode de signature :

  ```
  public static void main (String[])
  ```

  Eclipse ajoute automatiquement cette méthode si l'on coche la case adéquate lors de la création de la classe. Il suffit ensuite de remplir son corps :



Le but de cet exercice est de faire afficher un texte à l'écran (en réalité sur la console d'exécution *Eclipse* de notre application). Pour afficher un texte à l'écran, il faut utiliser la classe `System` (du package `java.lang` qui est importé

par défaut) et sa donnée membre **`out`** qui est une instance de la classe **`PrintStream`** (du package**`java.io`**).

La classe **`PrintStream`** contient tout un ensemble de méthodes **`print()`** et **`println()`** demandant l'écriture d'objets de différents types. Parmi elles, il y a la méthode **`println(String x)`** prenant en paramètre un objet de la classe **`java.lang.String`**.

Procédez comme suit :

1. Dans une nouvelle fenêtre du navigateur, afficher l'API de Java et consulter les classes **`java.lang.System`** et **`java.io.PrintStream`**
2. Complétez puis compilez et exécutez la classe exécutable **`Exo_01_01_01`** qui affiche le texte *Bonjour à tous !* Pour compiler et exécuter cette classe, il suffit d'aller dans le menu *Run*, puis *Run As* et choisir *Java Application*.
   Le texte s'affiche dans la console d'*Eclipse* (un onglet de la partie basse de la fenêtre).

Exo 1.1.2 - Utilisation d'un tableau

[Corrigé]

Dans l'onglet *Package Explorer*, sélectionner votre package **`exo_01_01`** et le copier-coller en nommant la copie **`exo_01_02`**. *Eclipse* y copie ainsi toutes les classes du package source et les adapte pour ce nouveau package.
Toujours dans cet onglet, développer le package **`exo_01_02`** et réaliser un clic droit sur la classe qu'il contient et sélectionner *Refactor* → *Rename*. Lui donner comme nouveau nom **`AuHasard`** et cliquer sur *Finish* en laissant les choix proposés par défaut. Passer outre le message d'avertissement. *Eclipse* se charge notamment d'adapter le code de la classe ainsi renommée à son nouveau nom.

Modifiez cette classe pour réaliser une application Java qui crée un tableau rempli avec *N* nombres entiers tirés « au hasard » dans l'intervalle [0..100[et les affiche à l'écran selon la présentation montrée ci-dessous. Ici, *N* est fixé à 10.

Exemple d'affichage à l'exécution :

```
[ 56 50 95 60 19 63 39 95 78 31 ]
```

Contrainte : l'affichage sera effectué dans une méthode statique **`afficher()`** distincte de la méthode **`main()`**. Cette méthode aura un unique argument, de type « tableau d'entiers ».

Indications :

- en Java, un tableau d'entiers est un **objet** de type **`int[]`** qui doit être créé avec l'opérateur **`new`**. Il possède une donnée membre **`length`** indiquant son nombre d'éléments. Pour en savoir plus, consulter dans la spécification du langage Java le chapitre consacré aux tableaux.
- il n'existe pas de variables globales en Java. Une variable est forcément une donnée membre d'un objet (par défaut) ou d'une classe (variable statique). Puisque nous n'avons *a priori* pas besoin de créer d'objet, nous stockerons la valeur de *N* dans une constante de classe déclarée ainsi :
  ```
  static final int N = 10;
  ```
- pour générer des nombres (pseudo-)aléatoires, cherchez parmi les méthodes **statiques** de la classe **`java.lang.Math`**, celle qui ne prend aucun paramètre et qui renvoie un **`double`** (un réel) entre 0.0 et 1.0 (non compris)
- en Java comme en C(++), on peut changer le type d'une expression, si cela a un sens. Si *x* est un **`double`**, alors on obtient un **`int`** (entier) éventuellement tronqué à partir de *x* par **`(int) x`**
- pour afficher un entier (type primitif) **`n`**, on peut appeler la méthode **`System.out.print(int i)`**. Or, celui-ci devant être suivi (ou précédé) d'un espace, il est plus confortable d'utiliser la conversion automatique en chaîne provoquée par l'opérateur de concaténation '**+**'. En effet, l'expression :
  ```
  n + " "
  ```
  provoque la conversion automatique de **`n`** en la chaîne (**`String`**) qui le représente. L'écriture précédente est alors transformée en :
  ```
  String.valueOf(n) + " "
  ```
  où **`String.valueOf()`** est une méthode statique de la classe **`String`** de profil :
  ```
  public static String valueOf(int i)
  ```

Remarque : l'exemple d'exécution ci-dessus est ce qui est écrit par le programme dans la console d'eclipse. Il n'y a pas la commande java qui permet l'exécution de la classe. Si la classe en question s'appelle **`Exo_01_01_02`** du

package **exo_01_02** du projet **TP1**, alors pour l'exécuter depuis la ligne de commandes, il faut taper, depuis le répertoire **bin** du projet **TP1** :

```
$ java exo_01_02.Exo_01_01_02
[ 56 50 95 60 19 63 39 95 78 31 ]
$
```

Exo 1.1.3 - Les arguments de la ligne de commande

[Corrigé]

Faire une copie du package précédent et la nommer **exo_01_03**. Modifier la (nouvelle) classe de ce nouveau package pour que la valeur de *N* ne soit plus fixée à l'avance mais soit donnée comme argument de la commande qui lance l'application.

Exemple :

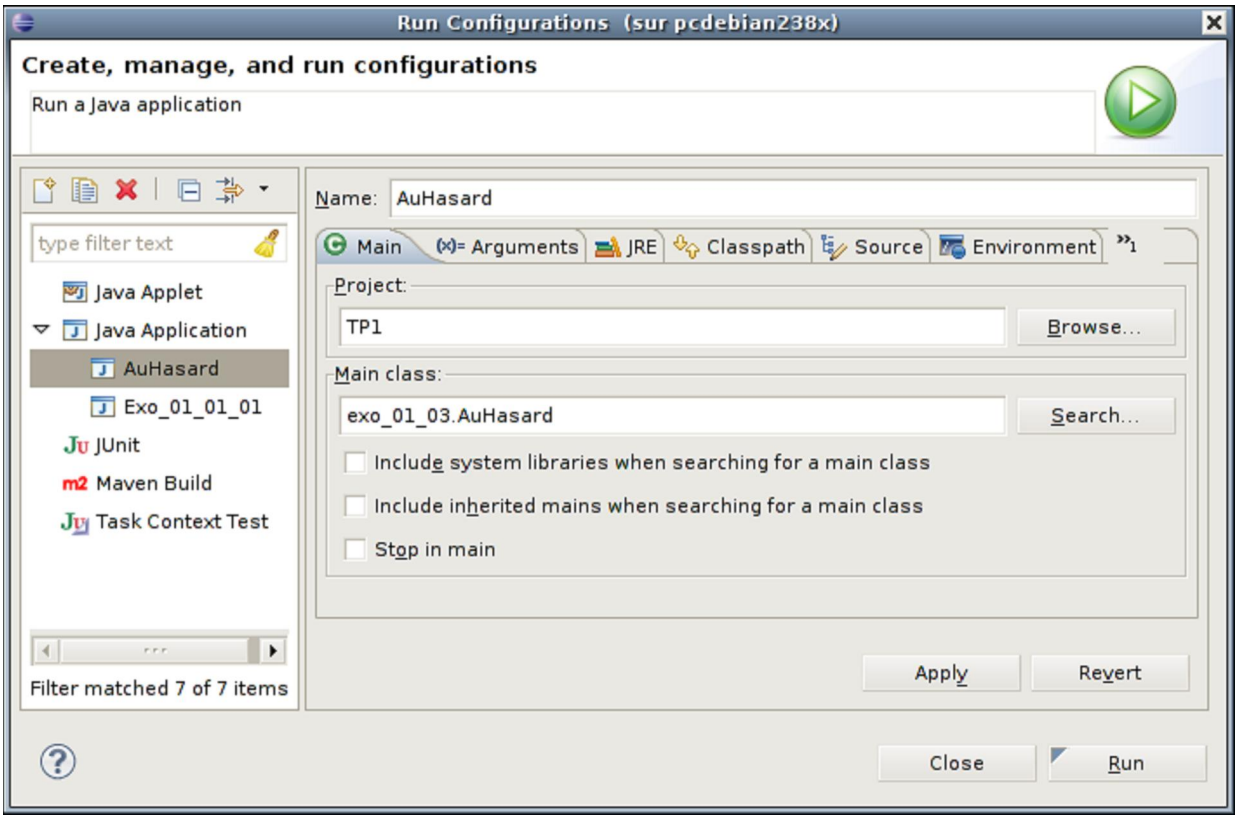
```
$ java AuHasard 20
[ 12 78 95 4 22 37 62 79 88 51 12 78 95 4 22 37 62 79 88 51 ]
$
```

Indications :

- les arguments de la ligne de commande sont stockés dans le tableau de **String** figurant dans le profil de la méthode **main()** :

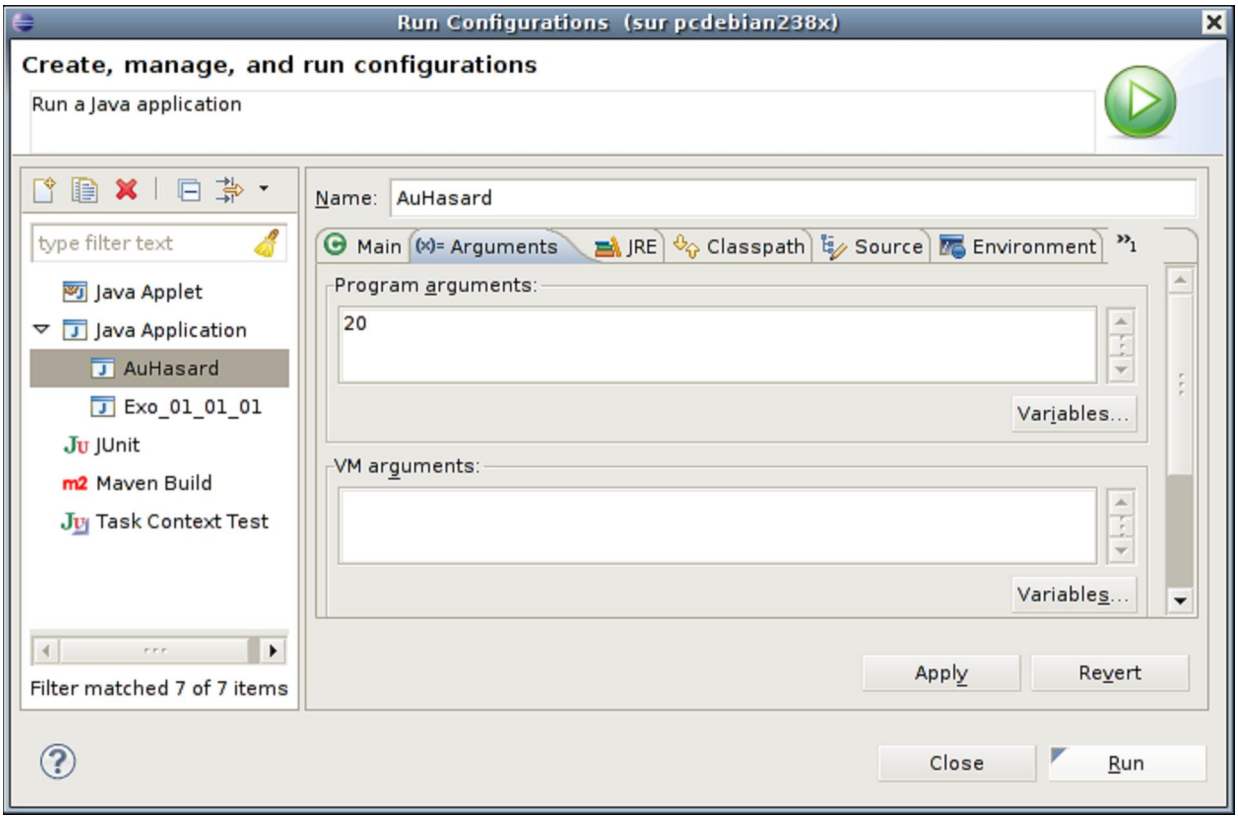
  ```
  public static void main(String[] args)
  ```

  Ici, le tableau est nommé **args** et **args[0]** contient le premier argument (la chaîne "20" dans l'exemple)
- sur la manière de convertir une chaîne en nombre, examinez les méthodes **statiques** de la classe **java.lang.Integer**. Attention cependant, cette classe fournit diverses méthodes statiques aux noms trompeurs. Ce que nous voulons, c'est obtenir un **int** (type primitif et non pas un objet **Integer**) à partir d'une chaîne (**String**).
- pour donner des arguments à une application dans *Eclipse*, il faut passer par le menu *Run* → *Run Configurations...* :
  - dans l'onglet *Main* de la fenêtre *Run Configurations*, il faut adapter la partie *Name* et la partie *Main class* (qui contiennent les informations sur la dernière classe exécutée) pour correspondre à la classe à exécuter et à son package :



  - dans l'onglet *Arguments*, on indique les arguments dans la partie *Program arguments* et on exécute

enfin en cliquant sur *Run* :



1.2 - Collections et itérateurs

Dans le même esprit que les exercices précédents, nous allons mémoriser des nombres de [0..100[tirés au hasard mais cette fois *N* **n'est pas connu à l'avance** : la mémorisation devra s'arrêter lorsque le nombre tiré est inférieur à 5, ce dernier nombre faisant lui aussi partie de la suite. Les tableaux n'étant plus adaptés, nous allons utiliser un objet d'une classe implémentant l'interface `Collection`, par exemple un objet `ArrayList`.

L'interface `Collection` est très générale et définit ce qu'une classe collectionnant des objets devrait fournir comme méthodes. Il n'y en a qu'une quinzaine parmi lesquelles :

- `add()` et `addAll()` pour ajouter des éléments
- `clear()`, `remove()`, `removeAll()` et `retainAll()` pour enlever des éléments
- `contains()`, `containsAll()` pour savoir si des éléments sont présents
- `isEmpty()`, `size()` pour connaître le nombre d'éléments
- `iterator()` pour obtenir un itérateur sur la collection (voir plus loin)
- ...

Elle est implémentée et étendue par les nombreuses collections spécialisées comme les ensembles (`Set`) qui garantissent qu'un élément est unique dans la collection, les listes (`List`) où les éléments ont des positions, et de nombreuses autres classes et interfaces.

Puisque nous voulons afficher les nombres dans l'ordre de leur tirage, nous avons besoin d'une classe qui maintienne cet ordre et qui permette de le suivre lors du parcours. C'est ce qu'impose l'interface `List` qui étend `Collection` notamment ainsi :

- les éléments sont indexés par un entier, le premier élément se trouvant à l'indice 0 ;
- la méthode `get()` est introduite et permet d'obtenir l'élément se trouvant à une position donnée ;
- les méthodes `add()` et `addAll()` ajoutent les éléments en fin de liste ou à une position donnée.

Nous avons tous les ingrédients pour résoudre notre problème. Il ne manque plus qu'à choisir une classe parmi celles implémentant `List`, par exemple `ArrayList`, `LinkedList`, `Vector` ... Dans les exercices qui suivent, nous choisirons `ArrayList`.

Exo 1.2.1 - Version « à l'ancienne »

[Corrigé]

Copiez à nouveau le package précédent et nommer la copie **`exo_02_01`**.

Examinez la documentation de **`java.util.Collection`**, **`java.util.List`** et **`java.util.ArrayList`**, en considérant dans un premier temps qu'on peut supprimer les notations **`<E>`** et en remplaçant **`E`** par **`Object`**. On se retrouve dans l'environnement de développement qui avait cours *avant* le JDK 1.5 (Java 5).

Dans ce contexte, les collections (et autres **`ArrayList`**) sont des collections d'**`Object`**. Un objet **`ArrayList`** peut être défini et initialisé par l'instruction :

```
ArrayList nombres = new ArrayList();
```

Nous allons faire mieux, en adoptant dès le début une bonne pratique de programmation orientée objet : pour déclarer une variable destinée à représenter des objets, utiliser comme type *l'interface la plus générale possédant les opérations souhaitées*. Puisque nous avons besoin de l'accès indexé, cela donnera ici :

```
List nombres = new ArrayList();
```

Pour que la ligne ci-dessus soit correctement écrite il faudra importer l'interface et la classe mentionnées en ajoutant au début du fichier source (après l'éventuelle instruction **`package ... ;`**) :

```
import java.util.List;
import java.util.ArrayList;
```

Pour ajouter un élément à la fin de la collection nous devons utiliser la méthode **`add()`** de profil (adapté pour notre contexte) :

```
public boolean add(Object e)
```

qui renvoie **`true`**, puisque dans le cas présent la collection est certainement modifiée par l'ajout d'un élément (il n'en aurait pas été de même si la collection avait été un objet **`Set`**).

Pour utiliser cette méthode, il nous faudra fabriquer un objet contenant le nombre tiré pour pouvoir le stocker dans la collection **`nombres`**. Le choix de la classe **`Integer`** s'impose. Si **`x`** est l'entier tiré, nous pourrons écrire simplement :

```
nombres.add(new Integer(x));
```

ce qui est parfaitement légal, puisque par **généralisation** (polymorphisme) un **`Integer`** est un **`Object`**.

La méthode **`afficher()`** pratiquera l'accès **indexé** aux éléments de la liste (c'est-à-dire qu'elle parcourra la liste comme si elle était un tableau). Pour cela, elle contiendra une simple boucle **`for`** avec un indice variant de 0 à **`nombres.size() - 1`**. L'élément d'indice **`i`** sera obtenu par appel de la méthode **`get()`** de profil (adapté pour notre contexte) :

```
public Object get(int index)
```

qui renvoie un **objet**. Si nous avions à manipuler l'**`Integer`** qu'il est en réalité et utiliser des membres de la classe **`Integer`** n'existant pas dans **`Object`**, nous serions amené à lui redonner son type par **particularisation** (polymorphisme) en « castant » le retour de **`get()`** :

```
Integer n = (Integer) nombres.get(i);
n.membreInexistantDansObject ...
```

Or, dans notre cas, nous n'avons qu'à écrire l'entier par appel de **`System.out.print()`**. En écrivant :

```
Object o = nombres.get(i);
System.out.print(o);
```

cette dernière instruction est équivalente à :

```
System.out.print(o.toString());
```

mais **`o`** étant en réalité un **`Integer`**, par particularisation automatique, c'est la méthode **`toString()`** de la classe

`Integer` qui sera utilisée et qui transformera correctement notre entier en chaîne. Le code, comprenant l'écriture de l'espace, peut alors être réduit ainsi :

```
System.out.print(nombres.get(i) + " ");
```

Exo 1.2.2 - Version à la mode Java 5

[Corrigé]

Continuons avec une nouvelle copie du package précédent nommée `exo_02_02`. Par la suite, cette indication ne sera plus forcément donnée...

Java 5 a introduit l'**emballage automatique** : lorsqu'une valeur d'un type primitif est mise là où il faut un objet, le compilateur se charge d'insérer la conversion adéquate (`int` en `Integer`, `double` en `Double`, etc.). Ainsi, la ligne :

```
nombres.add(new Integer(x));
```

peut s'écrire depuis Java 5 :

```
nombres.add(x);
```

Une importante avancée introduite dans Java 5 concerne les **classes et les types paramétrés** qui ont rendu les collections *génériques*. Le paramétrage des classes et types en Java joue un rôle analogue à celui des *templates* en C++. Reprenons à nouveau notre utilisation de `List` et `ArrayList` en tenant compte cette fois de la notation `<E>` et des types `E` :

- l'interface `List` et la classe `ArrayList` sont paramétrées. La seconde, par exemple, est déclarée ainsi (voyez la doc de l'API):
  ```
  public class ArrayList<E>
  extends AbstractList<E>
  implements List<E>, RandomAccess, Cloneable, Serializable
  ```
  il en découle que nous pouvons créer notre liste, ne pouvant contenir que des `Integer`, en paramétrant son constructeur :
  ```
  List<Integer> nombres = new ArrayList<Integer>();
  ```
- le compilateur contrôlera qu'il n'est ajouté que des `Integer` dans `nombres` puisque le profil de `add()` qui était :
  ```
  public boolean add(E e)
  ```
  devient :
  ```
  public boolean add(Integer e)
  ```
- D'autre part, on a la garantie que les objets obtenus par appel de `get()` sont des `Integer` puisque son profil devient :
  ```
  public Integer get(int index)
  ```

Vous n'avez donc pas plus de modification de code à apporter, si ce n'est adapter le profil de `afficher()` qui prend maintenant en paramètre un objet `List<Integer>`.

Exo 1.2.3 - Les itérateurs

[Corrigé]

Reprendre l'exercice précédent, mais ici la méthode `afficher` **doit** avoir un unique argument de type (paramétré) `java.util.Collection` et parcourir cette collection à l'aide d'un objet de type `java.util.Iterator`.

Indication. Examinez les interfaces `java.util.Collection` et `java.util.Iterator`. On notera notamment :

- qu'une `Collection<E>` est une extension de l'interface `Iterable<E>`, et doit donc fournir une méthode de signature `public Iterator<E> iterator()` qui renvoie un itérateur (`Iterator<E>`) permettant de parcourir la collection ;
- qu'un `Iterator<E>` est un objet permettant de parcourir un objet itérable. Pour cela, il fournit les méthodes :

- **`public boolean hasNext()`** qui indique si oui ou non il reste des éléments à parcourir
- **`public E next()`** qui renvoie le prochain élément non encore parcouru. Cet élément est un objet de type **`E`**, où **`E`** est le type qui paramètre la classe **`Iterator`** et doit être le même (ou une super classe) que le type paramétrant la collection.

Exo 1.2.4 - La boucle for améliorée

[Corrigé]

Autre nouveauté introduite par Java 5, la boucle **`for`** améliorée. Elle permet de remplacer des parcours de tableaux ou d'objets itérables par des instructions plus simples (mais pas forcément plus appropriées au traitement à opérer) :

- Si tableau est un tableau d'élements d'un certain type *TypeElement*, alors la boucle :

```
for (i = 0; i < tableau.length; i++)
    traiter(tableau[i]);
```

  peut s'écrire :

```
for (TypeElement elt : tableau)
    traiter(elt);
```

- De même, si **`liste`** est une classe implémentant l'interface **`java.lang.Iterable`**, alors la boucle :

```
for (Iterator<TypeElement> iter = liste.iterator(); iter.hasNext(); )
    traiter(iter.next());
```

  peut s'écrire :

```
for (TypeElement elt : liste)
    traiter(elt);
```

Reprendre le code de l'exercice précédent pour exploiter cette boucle.

La suite des TP...