

IN101 – TP7 2011/2012 v2
Sujet (à n'imprimer qu'une seule fois par poste de travail)

Durée : 3 h

1 OBJECTIFS

- Maîtriser les collections et les itérateurs
- Maîtriser les interfaces
- Maîtriser deux utilisations de la classe Scanner

2 TRAVAIL A REALISER

Nota : le travail demandé doit être terminé, en séance ou, à défaut, hors séance.

2.1 Créer un répertoire de travail

Si BlueJ n'est pas ouvert, le lancer. **Visualiser le sujet dans un navigateur** pour bénéficier des liens.

Créer un répertoire `tp7` dans `In101` sur votre compte. *C'est dans ce répertoire que devront être stockés tous les programmes Java et exercices relevant de ce tp.*

2.2 Exercice 1 : Projet "collection_de_formes" (interface, collection, itérateur)

Cet exercice s'inspire du projet `formes_pkg` du [tp6](#).

2.2.1 Ouvrir le projet

Télécharger le fichier [collection_de_formes.jar](#) lié à cet énoncé, et l'enregistrer dans le répertoire `tp7` précédemment créé.

Lancer *BlueJ* et ouvrir, le fichier `.jar` sauvegardé ci-dessus.

2.2.2 Parcourir le projet `collection_de_formes`

Ce projet correspond à peu près à ce à quoi vous deviez aboutir à la fin du TP6, mais en ayant remplacé la classe `Maison` par la classe `ArtModerne` (quasi-vide pour l'instant).

2.2.3 Générer une couleur aléatoirement

1.Écrire dans la classe `ArtModerne` une méthode d'instance `retUneCouleur()` qui devra retourner une couleur (`String`) aléatoirement parmi "red", "green", "blue".

a) Pour cela, utiliser **un objet de la classe** `Random` du JDK et un tableau des 3 couleurs possibles.

b) Pour éviter la création/destruction répétitive de ces 2 objets, les déclarer comme attributs.

L'un et/ou l'autre doit-il être une constante de classe ?

c) AIDE : **Consulter** la javadoc [version 6](#) de la méthode `nextInt(int)` de la classe `Random`.

2.**Essayer** la nouvelle fonction en affichant vingt fois dans la méthode `main()` le résultat de l'appel à `retUneCouleur()`.

3.**Tester** la nouvelle fonction à l'aide de la méthode `testRetUneCouleur()` de `ArtModerneTest`.

2.2.4 Générer une forme aléatoirement

1. **Écrire** dans la classe `ArtModerne` une fonction `retUneForme()` qui devra retourner une Forme aléatoirement parmi cercle, carré, triangle ou hexagone.
 - a. Elle prendra un seul paramètre de position (*celle-ci ne sera donc pas aléatoire*) qui sera utilisé à la fois pour les 2 coordonnées de la forme (*ce qui la placera forcément sur une diagonale*).
 - b. La couleur sera générée aléatoirement, ainsi que la (ou les) dimension(s) nécessaire(s) ; chaque dimension devra varier entre 15 et 45 inclus (**pas de boucle !**).
 - c. Pour cela, utiliser votre générateur aléatoire et une [structure de choix multiple](#).
Interdiction de créer plus d'une forme à chaque appel de `retUneForme()`.
(*ce qui est différent de `retUneCouleur()` où on avait créé les 3 String et choisi parmi les 3*)
2. **Essayer** la nouvelle fonction en l'appelant 10 fois dans la méthode `main()` (au lieu des appels à `retUneCouleur()`), en partant de (25,25) et avec des positions espacées de 25, puis en affichant les résultats par `S.o.println()`. On améliorera l'affichage à l'exercice suivant.
3. **Tester** cette fonction avec `testRetUneForme()` dans `ArtModerneTest`.

2.2.5 Afficher une représentation textuelle d'une forme

1. Les affichages précédents ne sont pas très satisfaisants. **Modifier** la fonction `toString()` dans `Forme` pour qu'elle ajoute le nom du type constaté au début de la String qu'elle retourne : `{classe:x,y,couleur}` par exemple : `{Cercle:75,75,green}`
2. Pour cela, **lire** la javadoc de `getClass()` ; ensuite, dans la classe qu'elle retourne, deux fonctions semblent **a priori** intéressantes : `getName()` et `getSimpleName()`.

2.2.6 Gérer une collection de formes

1. Dans la classe `ArtModerne`, **déclarer** un attribut `aListe` du type « liste de formes ». Pour rester le plus général possible, utiliser l'interface plutôt qu'une de ses implémentations.
2. **Modifier** le constructeur de la classe `ArtModerne` pour qu'il ajoute à la liste **qu'il aura créée** (choisir une `ArrayList`) `pNb` formes générées aléatoirement comme au 2.2.4.3., et remplacer le contenu de la méthode `main()` par un appel au constructeur en lui passant 10.

2.2.7 Afficher la collection de formes

1. **Créer** une procédure `afficheLesFormes()` qui devra afficher textuellement toutes les formes de la liste (**parcours simple** d'une collection) + *une ligne de --- dans le Terminal*.
2. Dans la méthode `dessine()`, **afficher** graphiquement toutes les formes de la liste (toujours un **parcours simple**), puis **appeler** la procédure créée à l'exercice 2.2.7.1.
3. A la fin de `main()`, **ajouter** un appel à `dessine()`.
4. **Essayer** plusieurs fois la méthode `main()` et vérifier la cohérence des affichages.

2.3 Exercice 2 : Le filtrage d'une collection

2.3.1 L'interface Condition

1. Pour pouvoir filtrer une collection, nous avons besoin de pouvoir passer en paramètre à la méthode de filtrage une condition que devra vérifier toute forme que l'on voudra supprimer de la collection.
Écrire l'interface `Condition` ne comportant qu'une fonction booléenne `verif()` avec un paramètre `Forme` (pour vérifier une certaine propriété de cette forme).
2. Pour pouvoir tester, il nous faut créer au moins une condition. **Écrire** la classe `EstUnHexagone` sans constructeur qui vérifie si la forme passée en paramètre à `verif()` est bien un hexagone.

2.3.2 Supprimer des formes

1. **Écrire** dans `ArtModerne` la procédure `supprimerSi()` qui accepte en paramètre une `Condition` et qui supprime de la liste toutes les formes qui vérifient cette condition (et ne les affiche plus).
Attention ! Ce n'est plus un parcours simple puisqu'il y a modification de la collection.
2. **Créer** une procédure `filtre()` appelant `supprimerSi()` avec la condition `EstUnHexagone`. Appeler cette nouvelle procédure à la fin du `main()`.
3. Pour pouvoir mieux visualiser ce qui se passe, **ajouter** au début de `supprimerSi()` une pause de 3 secondes (méthode de `Canvas`) et à la fin un appel à `afficheLesFormes()`.

2.3.3 D'autres conditions

1. **Écrire** une nouvelle condition `EstBleue` qui vérifie si la forme passée en paramètre est bien de couleur bleue.
2. **Écrire** une nouvelle condition `AUneSurfaceSuperieureA` qui vérifie si la forme passée en paramètre a bien une surface supérieure à celle stockée dans cette condition (et passée en paramètre au constructeur).
3. **Ajouter** à la fin de `filtre()` deux nouveaux appels à `supprimerSi()` : le premier pour supprimer les formes bleues, le second pour supprimer les formes de surface supérieure à 600.
5. **Essayer** plusieurs fois la méthode `main()` et vérifier les affichages graphiques et textuels.

2.4 Exercice 3 : Utiliser des algorithmes déjà programmés

2.4.1 L'inversion

1. Dans `dessine()`, **mettre** en commentaires (`ⓧ`) tout ce qui précède et tout ce qui suit l'appel à `afficheLesFormes()`.
2. Puisque `aListe` est une collection, nous pouvons lui appliquer tous les algorithmes fournis dans la classe `Collections`. **Créer** une procédure `algo()` appelant `reverse(collection)` puis à nouveau `afficheLesFormes()`. Appeler cette nouvelle procédure à la fin du `main()`.

2.4.2 Le mélange aléatoire

1. **Ajouter** maintenant à la fin de `algo()` un appel à `shuffle(collection)`, puis à nouveau à la méthode `afficheLesFormes()`.
2. **Essayer**.

2.4.3 Le tri

1. **Ajouter** maintenant à la fin de `algo()` un appel à `sort(collection)` sans paramètre, puis à nouveau à `afficheLesFormes()`. Pourquoi cela ne compile-t-il plus ?
2. L'erreur n'est pas facile à **interpréter**. Mais la javadoc de `sort(collection)` dit :
« All elements in the list must implement the `Comparable` interface ».
Ce n'est pas le cas des `Forme` ...
3. **Ajouter** ce qu'il faut dans `Forme` pour que les formes soient comparables par leur surface ;
aide : `compareTo()` est déjà définie dans la classe `Double` ...

2.5 Exercice 4 : La classe Scanner (travail personnel *)

Créer un nouveau projet BlueJ et une nouvelle classe.

2.5.1 Intéraction avec le clavier

1. Dans une méthode statique `test1()` :
 - **déclarer** (au moins) 2 variables entières `premier` et `second`
 - déclarer un `Scanner` et le connecter au clavier
 - déclarer une `String` `nom` et un entier `valeur`
2. **Faire** répétitivement (au moins une fois !) :
 - demander un nom de variable (affichage du message + saisie d'un mot)
 - demander une valeur (affichage du message + saisie d'un mot)
 - selon le nom saisi, affecter la valeur à la bonne variable (sauf si le nom n'existe pas)
 - afficher le nom et la valeur des (au moins) 2 variables
 - arrêter la boucle si le nom saisi était « fin »
3. **Compiler. Tester.** Que se passe-t-il si on saisit autre chose qu'un nombre quand on nous demande une valeur ? **Traiter** l'exception en affichant un message signalant qu'on attend une valeur et en n'oubliant pas que le mot « fautif » n'a toujours pas été lu ...

2.5.2 Analyse de String

1. Dans une méthode statique `test2()` :
 - **déclarer** (au moins) 2 variables entières `premier` et `second`
 - déclarer un `Scanner` et le « connecter » au clavier
 - déclarer une `String` `nom` et un entier `value`, ainsi qu'une `String` `ligne`
2. **Faire** répétitivement (au moins une fois !) :
 - demander un nom de variable suivi d'une valeur (affichage du message + saisie d'une ligne)
 - déclarer un `Scanner` et le « connecter » à la ligne précédemment lue
 - récupérer le nom et la valeur saisis **en testant préalablement leur disponibilité**
 - selon le nom saisi, affecter la valeur à la bonne variable (sauf si le nom n'existe pas)
 - afficher le nom et la valeur des (au moins) 2 variables
 - arrêter la boucle si le nom saisi était « fin »
3. **Compiler. Tester.** Que se passe-t-il si on ne saisit rien ou si on saisit autre chose qu'un nombre après le nom de variable ? C'est l'avantage des tests préalables ...

2.6 Terminer la séance

Si pas fait antérieurement, **générer** (après l'avoir complétée !) la documentation de ces deux projets, puis sauvegarder les projets ouverts, puis fermer BlueJ [menu Projet, choix Quitter]. Si besoin, envoyer par mél à votre binôme, en fichiers attachés, tous les projets de ce tp (exportés sous forme de fichiers `.jar`). Se déloger.

Ce sujet a été élaboré par Denis Bureau.

* **travail personnel** : doit être commencé au cours de la séance de TP s'il reste un peu de temps