Exercice 1.0 Le B et le A du « B A BA »

Mots-clés **:** *méthode main, package, fichier.*

Ecrire un programme Java qui affiche un message « Bonjour Polytech ! » sur la sortie standard. La classe que vous définirez appartiendra au package *com.polytechtours.di*.

Exercice 1.1 Prise en main de l'environnement Java hors de tout IDE.

1. Créez la classe de l'exercice 1.0 à l'aide d'un éditeur de texte quelconque. Compilez votre classe en ligne de commande, puis, toujours en ligne de commande, exécutez votre programme. Le compilateur (commande `javac`) et la machine virtuelle (commande `java`) sont accessibles dans le répertoire \bin\ de l'installation du JDK sur votre machine.
   Exemples d'appel standard :
   ```
   javac com\polytechtours\di\Test.java
   java com.polytechtours.di.Test
   ```
   NB : Commencez par mettre à jour les variables d'environnement requises…

2. Créez maintenant une archive JAR exécutable (commande `jar` avec les options `cfe`).
   Exécutez-la en ligne de commande (commande `java -jar`).

3. Ouvrir votre archive .jar avec IZARC ou tout autre outil manipulant les ZIP et étudiez sa structure.

Exercice 1.2 Prise en main d'un IDE : Eclipse.

1. Utilisez la classe de l'exercice 1.0 et suivez le tutoriel « officiel » Eclipse accessible via le lien suivant : http://www.vogella.de/articles/Eclipse/article.html. Mettez à jour les noms des projets/classes/packages en fonction des besoins de la cause (exercice 1.0).
   Les sections importantes dans un premier temps sont les sections 7 et 8.
   NB : Dans la partie 8.1, vous pouvez également exporter votre programme pour créer une archive exécutable (ne nécessitant pas l'utilisation de l'option `-classpath` à l'exécution…)

2. Tester maintenant l'exercice 1.3 (ci-dessous) dans un nouveau projet. Utilisez le **débugger** pour suivre le fonctionnement pas à pas des différentes versions de votre programme. (Vous pourrez vous aider du tutoriel Eclipse Debugging suivant : http://www.vogella.de/articles/EclipseDebugging/article.html)

3. Rajouter des commentaires Javadoc à vos classes, attributs et méthodes et générer la documentation html.

Exercice 1.3 Une première vraie classe

Voici le code source de la classe Livre :

```
public Class Livre {
  // Attributs
  private String titre, auteur, numISBN;
  private int nbPages
```

```
  // Constructeur
  public Livre Livre(String unAuteur, String unTitre) {
    auteur=unAuteur;
    this.titre=unTitre;
  }
  // Accesseurs
  public String getAuteur() {
    return auteur;
  }
  // Modificateurs
  void setNbPages(int nb) {
    nbPages=nb;
}
```

1. Corriger les éventuelles erreurs syntaxiques et ajouter une méthode `main` pour créer 2 livres et faire afficher les auteurs de ces deux livres.
2. Ajouter un accesseur pour les attributs `titre` et `nbPages`. Ajouter également un modificateur pour les attributs `auteur` et `titre`. Ajouter enfin un accesseur et un modificateur pour l'attribut `numISBN`.
3. Sachant que `"Freud"` est équivalent à `new String("Freud")`, si on modifie la méthode `main()` de la façon suivante, l'auteur du livre est-il modifié ? Pourquoi ?

```
public static void main(String[] args){
      Livre livre1;
      livre1 = new Livre("Rabelais","Gargantua");
      String unAuteur = livre1.getAuteur();
      unAuteur = "Freud" ;
}
```

4. Dans le fichier `Auteur.java` (du même package), définir une nouvelle classe `Auteur` contenant 3 attributs de type `String` : `nom`, `prénom` et `nationalité`. Compléter cette classe avec constructeurs, accesseurs et modificateurs publics, et modifier la classe `Livre` pour qu'elle l'utilise.

5. Que se passe-t-il si on modifie la méthode `main(…)` de la façon suivante (en supposant que les constructeurs utilisés existent et sont corrects) :

```
public static void main(String[] args){
      Livre livre1 ;
      livre1 = new Livre(new Auteur("Rabelais"),"Gargantua");
      Auteur unAuteur = livre1.getAuteur();
      unAuteur.setNom("Freud");
}
```

6. Et si on remplace maintenant la dernière ligne du `main` précédent par la ligne suivante ? Pourquoi ?

```
      unAuteur = new Auteur("Freud");
```

7. Dans la méthode `main`, donnez une valeur au nombre de pages de chacun des deux livres, faîtes ensuite afficher ces nombre de pages et enfin calculer le nombre total de pages de ces deux livres et affichez-le sous la forme suivante :

   `Le nombre de pages total est : 643`

8. Changer le modificateur de `nbPages` : il ne devra changer le nombre de pages que si on lui passe en paramètre un nombre positif, sinon il renvoie une exception.

QUESTIONS BONUS : *(Les questions suivantes sont facultatives…)*

9. *Ajouter une méthode publique `toString()` qui renvoie une chaîne de caractères qui décrit le livre. Elle redéfinit celle fournie par la classe `Object` dont le prototype est `public String toString()`.*
10. *Ajouter un prix (décimal !) au livre et les accesseur/modificateur correspondants. Ensuite, ajouter au moins un constructeur qui permet d'initialiser le prix. Modifier la description du livre retournée par `toString()` afin qu'elle fasse apparaître « Prix pas encore fixé » si ce prix n'a pas été initialisé.*

11. *On souhaite bloquer le numéro ISBN de sorte qu'une fois fixé il ne puisse plus être modifié. Pour faire cela, vous utiliserez un attribut booléen « numISBNFixed ». Ecrire l'accesseur associé.*
12. *Modifier le modificateur du nombre de pages pour qu'il affiche aussi un message sur la sortie standard d'erreur (en plus de lever une exception).*
13. *Rajouter un attribut genre qui prendra ses valeurs dans {ROMAN, BIOGRAPHIE, ESSAI} et un attribut style qui prendra ses valeurs dans {AMOUR, POLICIER, AVENTURE, SCIENCE FICTION}.*

Exercice 1.4 L'héritage chez les animaux.

Les hommes sont des mammifères. Les mammifères sont des animaux. On considère que tout Animal, quel qu'il soit, a un nom (« Jean-Martin » ou « Rex » par exemple) et un sexe. Vous modéliserez le sexe sous la forme d'un attribut de type *enum* (type que vous définirez dans un fichier séparé). Le sexe sera soit MASCULIN, soit FEMININ, soit NEUTRE.

Question 1 : Ecrivez 3 classes Java pour représenter cette hiérarchie.

Ajoutez aux mammifères un attribut entier pour définir le nombre de mamelles et un attribut `String` pour désigner l'espèce, et aux hommes un attribut `String` pour le numéro de sécurité sociale et un second pour l'adresse postale.

Toutes vos classes devront comporter le constructeur prenant en paramètre le nom de l'animal.

Question 2 : Connaissant le comportement « par défaut » des constructeurs en Java (cf. encadré ci-dessous), vérifiez que vos constructeurs se comportent correctement. Pour cela, vous dessinerez l'arbre d'appel des différents constructeurs de vos classes, en remontant jusqu'à la classe `Object`. Représentez pour commencer le chaînage par défaut (si vous n'ajoutez aucune instruction `super(...)` ni aucune instruction `this(...)`) puis, d'une autre couleur, le chaînage que vous désirez obtenir. Comment alors s'y prendre pour obtenir le chaînage souhaité ?
NB : On rappelle que de manière générale un attribut xxx d'une classe YYY n'a aucune raison d'être initialisé dans le corps d'une classe fille de YYY. Cette initialisation doit être prise en charge par la classe YYY elle-même !

- `super(…)` permet d'appeler un constructeur de la classe mère,
- `this(…)` permet d'appeler un autre constructeur de la même classe,
- `super(…)` et `this(…)` ne sont utilisables qu'en tant que 1ère instruction d'un constructeur,
- si la 1ère instruction d'un constructeur n'est ni `super(…)` ni `this(…)`, java insère automatiquement et systématiquement l'instruction `super()` ;
- si aucun constructeur n'est défini pour une classe donnée, java crée implicitement le constructeur sans argument, qui ne fait qu'appeler `super()` ;

Question 3 : Quel(s) constructeur(s) faut-il modifier pour afficher un message (« Une bébête est créée ! ») sur la sortie standard à chaque fois qu'une instance d'une de vos classes est créée ? Effectuer cette(ces) modification(s).
Rajouter un compteur du nombre d'instances créées pour chaque classe. Peut-on le décrémenter correctement ?

Question 4 : Voici ci-dessous une classe de test et l'affichage que vos classes devront fournir. Pour ce faire, la méthode `toString()` de vos classes devra fournir une description de l'instance, en incluant les descriptions de toutes les classes ancêtres (vous utiliserez le mot-clé `super` pour appeler une méthode définie dans la classe mère et non la méthode de la classe en cours qui la redéfinit).

```
public class TestAnimal {
    public static void main(String[] args){
        Animal[] animaux = new Animal[5];
        animaux[0] = new Mammifere("MachinTruc");
        animaux[1] = new Mammifere();
        animaux[2] = new Animal("Médor");
        animaux[3] = new Homme();
        animaux[4] = new Homme("Robert");
        for(int i=0;i<animaux.length;i++)
            System.out.println(animaux[i]);
    }
}
```

L'affichage obtenu est alors :

```
Je suis un animal de nom MachinTruc. Je suis un mammifère.
Je suis un animal. Je suis un mammifère.
Je suis un animal de nom Médor.
Je suis un animal. Je suis un mammifère. Je suis un homme.
Je suis un animal de nom Robert. Je suis un mammifère. Je suis un homme.
```

QUESTIONS BONUS : *(Les questions suivantes sont facultatives…)*

Question 5 : Dans l'exercice 1.2, rajouter une classe Chien (mammifère, donc), une classe Poisson et une classe Carpe (c'est un poisson lol) ayant chacun des attributs spécifiques Ecrire correctement les constructeurs ainsi que les accesseurs et modificateurs des attributs. Redéfinissez également `toString()` conformément à la question 4 ci-dessus.

Exercice 1.A MesMaths

Vous allez rédiger une classe `MesMaths` permettant de fournir des primitives de calcul mathématique. Vous gèrerez correctement les cas d'erreur avec des exceptions.
Question 1 : Ecrire une méthode `factorielle` qui prend en paramètre un entier et qui renvoie son factoriel. (*Rappel : factorielle(i)=1*2*3*….*(i-1)*i).*
Question 2 : Ecrire une méthode `puissance` qui prend en paramètre deux entiers i et j et qui renvoie la valeur de i élevé à la puissance j. (*Rappel : i^j=i*i*i….*i (j fois)).*
Question 3 : Ecrire une méthode `puissanceDe10` qui prend en paramètre un entier et qui renvoie la puissance de 10 associée. (*Rappel : 10^k=10*10*…*10 (k fois)).*

Exercice 1.B Factorielle

On se propose de créer une classe Java qui calcule le factoriel d'un nombre qu'on lui fournit dans son constructeur. Dans ce but, tester la classe suivante.

Pourquoi le factoriel de 5 vaut-il 1 ici ?

Utilisez le mode pas à pas du débugger pour identifier ce qui se passe…

```
public class Factorielle {
  int valeur;
  public void Factorielle() {
    this.valeur = 5;
  }
  public int getResultat() {
    int i, resultat = 1;
    if (valeur > 1)
      for (i = 1; i <= valeur; i++)
        resultat *= i;
    return resultat;
  }
  public static void main(String[] args) {
    System.out.println("Resultat : "
              +(new Factorielle()).getResultat());
  }
}
```

Exercice 1.C Animaux Lvl.2

Définir la classe Animal comme une classe abstraite. Faire les modifications nécessaires pour que tout compile.
Modifier la classe Animal : pour gérer son affichage, il aura également un attribut définissant sa couleur (classe `java.awt.Color`) et 4 attributs de type `int` : posX, posY, sizeX et sizeY. La classe `Animal` contiendra également une méthode abstraite `void dessiner(Graphics g)` qui sera utilisée pour spécifier comment l'animal……se dessine (on l'utilisera plus tard, on verra comment…). *(NB : je vous laisser chercher à quel package la classe `Graphics` appartient…)*

Exercice 1.D MyLightInteger et MyLightFloat

Le package `java.lang` contient notamment les classes `Integer`, `Float`, … qui englobent des types primitifs sous forme de classes (les « *wrapper class* »). Nous allons réécrire nos propres versions de ces classes en ne fournissant qu'un sous-ensemble de services.

Question 1 : En partant de la documentation de la classe `Integer`, écrire une classe `MyLightInteger` qui implémente seulement les fonctionnalités suivantes :

- Les 3 constantes MAX_VALUE, MIN_VALUE et SIZE,
- Les deux constructeurs,
- Les méthodes de conversion d'`int` et d'`Integer` en `String`, et inversement, à l'exception de celles qui ne fonctionnent pas en base dix, *i.e.* qui travaillent avec un 2ème paramètre « `int radix` » : 5 méthodes sont à écrire.

Bien sûr, vous n'avez pas le droit d'utiliser la classe `Integer` dans votre classe...

Cf. http://download.oracle.com/javase/6/docs/api/index.html?java/lang/Integer.html
…et http://download.oracle.com/javase/6/docs/api/constant-values.html#java.lang.Integer.MAX_VALUE pour les valeurs des constantes.

Question 2 : Faire la même chose pour la classe `Float`.

Exercice 1.E MySystem

Ecrire une classe `MySystem` telle que « `MySystem.println(XXX)` » fait :

- La même chose que `System.out.println(XXX)` si l'objet XXX est « `Printable` »
- Lève une exception « `NonPrintableObjectException` » sinon.

Attention ! Pour faire fonctionner votre classe comme la classe `System`, il vous faut écrire plusieurs méthodes `println(…)`…

Exercice 2.1 Hello

On désire réaliser une petite application graphique composée d'une unique fenêtre affichant 2 boutons. A chaque fois qu'il est cliqué, le premier bouton (« Appuie ! ») affiche sur la sortie standard le nombre total de fois où l'utilisateur a cliqué dessus depuis le lancement de l'application. Le second bouton (« Quitter ») ferme la fenêtre et quitte l'application.

L'application devra avoir à peu près cette allure :

Question 1 : Créer le squelette de l'application : une classe et la fonction main qui instancie une instance de cette classe. Notez qu'à partir de maintenant, sauf question explicite, plus rien de sera ajouté dans la fonction main. Pensez à réaliser les `import` dont vous allez inévitablement avoir besoin.
Question 2 : Pour pouvoir effectuer ce qu'on lui demande, la classe devra comporter au minimum un certain nombre d'attributs. Lesquels ? Les ajouter dans la définition de la classe.
Question 3 : Il faut maintenant faire en sorte que les éléments qui doivent être affichés le soient. Pour cela, un certain nombre d'instructions doivent être positionnées quelque part dans votre code. Quelles sont ces instructions ? Où faut-il les positionner ? Le faire. Attention : il s'agit uniquement ici de gérer l'affichage, pas la gestion événementielle.
Indications : Trois types d'opérations sont nécessaires.
Question 4 : Mettre en œuvre la gestion des événements pour faire en sorte que :

- l'appui sur le bouton « Quitter » quitte l'application,

- l'appui sur le bouton « Appuie ! » affiche sur la sortie standard le nombre total de fois où l'utilisateur a cliqué dessus depuis le lancement de l'application.

Indications :

a) Pour chaque bouton, il y a a priori deux étapes, qui peuvent être réalisées en une seule.

b) L'interface `ActionListener` contenant uniquement la méthode `void actionPerformed(ActionEvent e)` pourra (!!??) vous être utile.

Question 4 bis : Quelles sont les 4 façons de créer un `Listener` ?
Indication : la première consiste à utiliser une classe anonyme…
Question 5 : Que se passerait-il si on ajoutait une commande `System.out.println("hello");` dans la fonction main, après la seule commande qui y est pour l'instant ?
Question 6 : On veut maintenant que le clic sur le bouton « Appuie ! » n'entraîne plus d'affichage sur la sortie standard mais plutôt un affichage dans la fenêtre de l'application (par exemple un label entre les deux boutons). On continue d'afficher le nombre total de clic. Comment faire ? Le faire.
Attention à afficher votre label correctement, y compris quand le nombre de clics atteint 10 (ou 100…)
Question 7 : Créez une archive JAR exécutable de votre application et tester son exécution.

Supplément gratuit : Voici quelques commandes qui pourraient vous être utiles…

- *`setBorder(new EmptyBorder(int,int,int,int));` (pour un `JComponent`) permet d'ajouter un "bord" vide autour d'un composant (tailles en pixels : haut, gauche, bas, droite)… particulièrement utile pour ne pas trop alourdir les interfaces quand on utilise pack().*
- *`setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);` (pour un `JFrame`) force l'application à se terminer dès que la fenêtre est fermée (par exemple lors d'un clic sur la croix en haut).*
- *`setLocationRelativeTo(null);` (pour un `JFrame`) positionne la fenêtre au centre de l'écran.*
- *`setResizable(boolean);` (pour un `JFrame`) définit si oui ou non la fenêtre peut être redimensionnée par l'utilisateur.*

Exercice 2.2 Copie de fichiers

1. Ecrire un programme pour permettre à l'utilisateur de faire la copie d'un fichier. Pour cela, il spécifie le nom du fichier à copier (chemin inclus) ainsi que le nom et l'emplacement du fichier destination. Un bouton permet de lancer la copie.
   Si le fichier destination existe déjà prendre la précaution de demander à l'utilisateur s'il veut l'écraser ou non.
   Indications :
   - *Utiliser la classe `JFileChooser` pour rendre votre appli plus pratique…*
   - *Si vous utilisez JAVA SE 6, la lecture octet par octet du fichier source est nécessaire via les classes de java.io.*
   - *Si vous utilisez JAVA SE 7, l'utilisation du package java.nio.file devrait vous simplifier la vie…*

QUESTIONS BONUS :

2. Modifier votre programme pour qu'il fonctionne à la fois via une interface graphique ET en ligne commande.
   Usage en ligne de commande :
   « `myProg` » lance le programme avec son interface graphique,
   « `myProg SourceFilePath DestinationFilePath` » fait la copie de Source vers Destination sans l'interface graphique.

3. Mettre à jour votre programme pour que :
   a. La copie se fasse avec le même nom de fichier si le chemin Destination correspond à un nom de répertoire existant.
   b. La copie s'applique sur tous les fichiers d'un répertoire si le chemin Source correspond à un nom de répertoire existant.

Exercice 2.3 Calculette

1. Créer une petite calculatrice pour effectuer les 4 opérations standards : addition, soustraction, multiplication et division.
2. Faites en sorte qu'un message s'affiche sur l'interface si l'utilisateur effectue des saisies erronées (exemple : division par zéro ou nombre mal formé). *Vous pourrez par exemple utiliser une des méthodes statiques de la classe* `JOptionPane`*).*

Exercice 2.A SMS

Ecrire un programme qui simule la rédaction d'un SMS : une zone de texte permet de saisir le message à envoyer (maximum : 160 caractères). A tout instant, le nombre de caractères restant est affiché à l'utilisateur. Un premier bouton permet d'envoyer le message (qu'on simule par un affichage sur la sortie standard), un second de réinitialiser la zone de texte et un dernier de quitter.

Exercice 2.B Hello Lvl.2

1. Dans l'exercice 2.1, rajouter deux compteurs qui comptent le nombre de fois où la souris rentre et sort du bouton « Appuie ! ». Afficher leurs valeurs dans la fenêtre.
2. Modifier le programme pour que chaque appui sur le bouton ouvre une nouvelle boite de dialogue (`JDialog`) qui affiche le nombre de clics... Un bouton ok permet de fermer cette boite de dialogue.
3. En repartant du résultat de la question 1, on veut maintenant 2 fenêtres identiques… telles que les clics sur le bouton « Appuie ! » de l'une d'elles affiche le nombre de clics sur la 2e (et inversement).
   Pour quitter l'appli il faut fermer les 2 fenêtres.

Exercice 2.C Date formatée

1. Ecrire un programme qui affiche la date et l'heure courante (en français☺) joliment formatée de la façon suivante :

   Nous sommes le mardi 10 avril 2007, il est 17h06 et 55 secondes.

Indication : les classes `java.util.Date` *et* `java.text.SimpleDateFormat` *devraient vous être utiles ainsi que – peut-être – la constante statique* `java.util.Locale.FRANCE`*.*

2. Mettre en place la gestion événementielle pour que la date soit mise à jour à chaque fois que la souris passe au-dessus du label affichant cette date.

Exercice 2.D Copie de fichiers Lvl.2

Modifier le programme de l'exercice 2.2 pour pouvoir effectuer plusieurs copies du fichier spécifié par l'utilisateur. Par exemple, les noms des 10 différentes copies d'un fichier « ABCD.EFG » s'appelleront ABCD-1.EFG, ABCD-2.EFG, …, ABCD-10.EFG
L'utilisateur choisit évidemment le nombre de copies à faire…

Exercice 2.E Animaux Lvl.3

On désire fournir à l'utilisateur un petit outil lui permettant de créer l'`Animal` de son choix (par exemple un `Homme` nommé Jules d'espèce Homo Sapiens Sapiens, habitant au 18 rue des mules, de N° de sécu 1234567, ou bien un `Mammifère` nommé Médor…… *etc.*). Pour cela, on construira une boite de dialogue dans laquelle il pourra spécifier le type d'`Animal` qu'il veut créer, et les différents attributs de cet `Animal`. Naturellement, il aura des champs différents à remplir en fonction du type d'`Animal` qu'il aura choisi...
Pour cela, vous pourrez utiliser les classes `JDialog`, `JRadioButton` et `ButtonGroup`.
Un bouton « ok » permettra de fermer la boite de dialogue après avoir construit l'instance de la classe dérivant d'`Animal` qui va bien.
(Dans une utilisation future éventuelle de cette classe nous modifierons l'action effectuée par ce bouton pour pouvoir utiliser en retour l'instance de l'objet nouvellement créé).

Exercice 2.F Morpion… tricheur

1. Réalisez une application *morpion*.
   Comme dans un morpion classique (cf. http://fr.wikipedia.org/wiki/Morpion_(jeu)), l'utilisateur et l'ordinateur jouent à tour de rôle. Quand c'est à l'ordinateur de jouer, il coche au hasard une case libre parmi les 9 cases du damier 3x3 (utiliser par exemple un damier de boutons). Quand c'est à l'utilisateur de jouer, il joue la case libre qu'il désire. Dès qu'une case est cochée par l'utilisateur, une croix X s'y affiche. Dès qu'une case est cochée par l'ordinateur, un rond O s'y affiche. Le gagnant est le premier du joueur ou de l'ordinateur qui arrive à aligner 3 croix (ou trois ronds), que ce soit horizontalement, verticalement ou en diagonale. L'ordinateur annonce alors qui a gagné en affichant d'une manière ou d'une autre un message après quoi le joueur peut recommencer une nouvelle partie s'il le désire. L'ordinateur sauvegarde et affiche en permanence le nombre de victoires du joueur et de l'ordinateur depuis le lancement de l'application, ainsi que le nombre de victoire consécutives du dernier vainqueur.
2. Là où ça se complique… c'est que l'ordinateur triche pendant le jeu de la façon suivante. Dès que le joueur s'apprête à effectuer sur une case un clic qui le ferait gagner, les cases changent de position dans le damier de manière à l'empêcher de gagner. Plus exactement, cette action de l'ordinateur est faite dès que la souris entre sur une case sur laquelle un clic ferait gagner l'utilisateur. Chaque case passe alors à la position symétrique par rapport à la case centrale (et donc la case centrale ne change pas de position, ce qui laisse à l'utilisateur la possibilité de gagner – parfois…).

Exercice 3.1 Prise en main des Threads

1. Ecrire une application Java qui affiche deux messages dans deux endroits différents (un en haut un en bas par exemple) d'une fenêtre/boîte de dialogue : « Je suis ici… » et « Maintenant je suis là ! ». Les deux textes s'affichent alternativement l'un après l'autre : toutes les demi-secondes, le message affiché disparaît tandis que l'autre apparaît. Cette « animation » sera gérée par un thread (un seul est nécessaire, en plus du Thread principal).
   NB : Prenez le réflexe de fournir dès maintenant à votre application tous les atouts d'une IHM réussie : `setResizable(false)` *si c'est une* `JFrame`*, réglages du comportement par défaut en cas de fermeture de la fenêtre, etc.*
2. Rajoutez maintenant un bouton STOP qui permet de stopper l'animation, ainsi qu'un bouton START qui permet de la lancer/relancer. STOP (resp. START) ne sera cliquable que si START (resp. STOP) est le dernier bouton à avoir été cliqué.
   Rappels : un thread ne peut être lancé (`start()`*) qu'une seule fois. Si vous avez besoin de « relancer » un thread, il faut en faire un nouveau.*

Exercice 3.A Un label clignotant.

Ecrire une classe autonome `JLabelQuiClignote` qui représente un `JLabel`… qu'on peut faire clignoter. On lancera le clignotement du label en appelant sa méthode lancerClignotement() et on l'arrêtera en appelant `arreterClignotement()`. Par défaut, le label ne clignote pas, c'est-à-dire qu'il faut appeler `lancerClignotement()` au moins une fois.

Exercice 3.B Chronomètre

Ecrire une application « chronomètre » qui affiche le temps écoulé depuis le lancement de l'application sous la forme « `hh:mm:ss` ». Trois boutons permettent de mettre en pause, lancer/relancer et réinitialiser le chronomètre. Le lancement du chrono est automatique au lancement de l'appli.

Exercice 3.C Threads Lvl.2

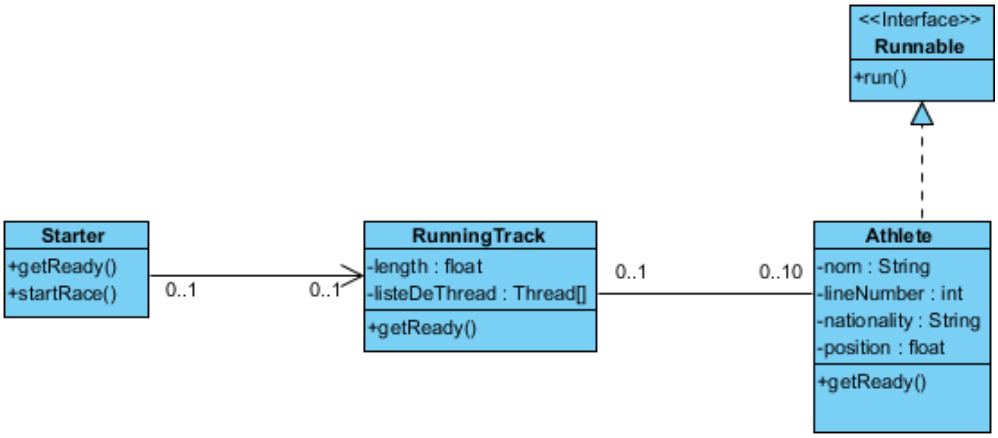
Modifier l'exercice 3.1 pour autoriser plusieurs clics successifs sur le bouton START ou sur le bouton STOP. Dans ce cas, chaque nouvel appui sur START lance un nouveau Thread qui fait la même chose que le Thread initial (Plusieurs Threads animerons donc simultanément l'affichage des textes…). De même, chaque nouvel appui sur STOP stoppe un des threads lancés.

Exercice 4.1 Mise en œuvre du *design pattern* Observer

L'objectif final de cet exercice est de vous faire modéliser puis implémenter un programma java à l'aide d'un *design pattern* simple (le D.P. Observer).
Pour cela, vous allez réaliser un programme java qui simule une course d'athlétisme. Dix athlètes participent à cette course. Chaque athlète sera une instance d'une classe `Athlete` à écrire. Le mouvement de chaque athlète sera réalisé par un Thread. Elle contiendra une méthode `public void run()` qui sera principalement un boucle dans laquelle l'athlète augmente sa position de 1 et s'endort.

Dans un premier temps vous allez réaliser ce programme sans interface graphique. Vous ferez ensuite une IHM swing utilisant un `Canvas`.

Voici un exemple de diagramme de classes (partiel) dont vous pourrez vous inspirer :



Le `Starter` joue le rôle de `main`. Il va instancier les différents objets nécessaires à la simulation. Le `RunningTrack` est la piste. C'est elle qui gère les Threads qui vont « animer » les athlètes.

1. Complétez ce diagramme de classes, puis créez les classes `RunningTrack` et `Athlete`. Créez ensuite la classe `Starter` chargée de construire et lancer la course. Vous initialiserez les 10 Threads avec des priorités aléatoires.
   Avant de poursuivre, assurez-vous que votre simulation s'exécute correctement (via le débugger ou via des mouchards type `System.out.println`…)
2. En appliquant le Design Pattern **Observer**, rédigez une classe `SimpleObserver` qui, pour chaque athlète, affiche sur la sortie standard la position de celui-ci chaque fois qu'il a bougé. Affichez également son nom et son numéro de ligne.
   Vous utiliserez pour cela l'interface `java.util.Observer` et la classe abstraite `java.util.Observable`. Commencez par mettre à jour le diagramme de classes !

Ce 1er `Observer` constitue une première vue (au sens Modèle-Vue ou Modèle-Vue-Contrôleur). Vous allez maintenant en créer une deuxième dans une IHM swing en vous appuyant sur un `Canvas`.
(to be continued)