TD 3 – POO (Ruby)

1) Des Documents

Pour la gestion d'une bibliothèque, on nous demande d'écrire une application manipulant des documents de nature diverse : des livres, des dictionnaires, etc.
Tous les documents ont un numéro d'enregistrement et un titre. Les livres ont, en plus, un auteur et un nombre de pages, les dictionnaires ont une langue et un nombre d'articles. Ces diverses sortes de choses doivent pouvoir être manipulées de façon homogène en tant que documents.

A) Définissez en Ruby les classes **Document**, **Livre** et **Dictionnaire**. On souhaite pouvoir créer des documents, livres et dictionnaires en fournissant les paramètres nécessaires à l'initialisation de leurs variables d'instances.

Exemple :

```
doc=Document.nouveau("Règlement Intérieur",1)
liv=Livre.nouveau ("Ruby pour les Nul",2,"Jaco",435)
dic=Dictionnaire.nouveau ("Le dico de la langue",8,"Ruby",652)
```

B) En fait il est inutile de fournir à la création des objets le numéro d'enregistrement, il serait bien plus utile de le faire générer par le système. Quelle solution proposez vous pour mettre cela en place ? Modifiez les classes dans ce sens.

Exemple :

```
doc=Document.Nouveau("Règlement Intérieur")
liv=Livre.nouveau ("Ruby pour les Nul","Jaco",435)
dic=Dictionnaire.nouveau ("Le dico de la langue","Ruby",652)
```

C) On souhaite pouvoir afficher toutes les informations relatives aux documents, livres et dictionnaire, modifier les classes en conséquences

```
Document : Numéro = 1, Titre = Réglement Intérieur,
Livre : Numéro = 2, Titre = Ruby pour les Nul, Auteur = Jaco, Nombre de pages=43
Dictionnaire : Numéro = 3, Titre = Le dico de la langue, Langue = Ruby, Nombre d'Articles = 652
```

D) Définissez une classe **Bibliothèque** réduite à une méthode permettant de tester les classes précédentes.

2) Les Comptes en Banque Episode 3

A partir de la classe Compte du cours 2, nous définissons deux sous-classes : la classe **CompteEpargne** et la classe **CompteCourant**.

- **Un compte courant** est un compte sur lequel des découverts sont autorisés.
- Un **compte épargne** est un compte qui rapporte des intérêts mensuels.

3) Simulation d'afficheurs lumineux

Le but de cet exercice est de simuler en Ruby les afficheurs lumineux qu'on voit un peu partout et qui font circuler un texte en boucle. Des exemples d'utilisations sont donnés en Annexes et votre code DOIT permettre de faire fonctionner ces exemples de la même façon.

A) Le décaleur

Intéressons-nous d'abord au *décaleur*. C'est un objet qui stocke une suite de L caractères avec L > 0 (et pas plus). L est constant et est appelé *largeur* du décaleur. Initialement, un décaleur contient L espaces.

Les quatre fonctionnalités d'un décaleur sont :

- `getLargeur` renvoie la largeur du décaleur,
- `raz` force tous les caractères à *espace*,

- `decale` décalage d'une position vers la gauche de sa suite de caractères : le caractère le plus à gauche est supprimé de la suite et est renvoyé par la méthode. Le nouveau caractère le plus à droite est fixé à la valeur du paramètre de la méthode.
- `to_s` renvoie sous forme de chaîne une copie du contenu de la suite de caractères du décaleur.

Définir entièrement en Ruby la classe `Decaleur`. Préciser en particulier ses variables d'instances et ce qu'elles représenteront. Écrire en Ruby le code de toutes les méthodes de la classe, sans oublier en particulier les méthodes de création/initialisation. On donne en Annexe 1 un rappel des méthodes de la classe `Array` de Ruby et en Annexe 2 un exemple d'utilisation de cette classe `Decaleur`

B) Les afficheurs lumineux

Les caractéristiques d'un afficheur lumineux sont les suivantes : il ne peut visualiser simultanément qu'un nombre N fixe et entier de caractères avec N > 0. Le message qui se déroule en boucle sur l'afficheur ne doit pas avoir une longueur nulle, en revanche, celle-ci peut être plus petite, égale ou supérieure à N .

Le message défile dans l'afficheur en se décalant d'une position vers la gauche à chaque top d'une horloge. Quand le dernier caractère du message vient juste d'entrer dans la partie visualisée, au prochain top horloge, c'est le premier caractère du message qui y entre à son tour.

L'interface de la classe `Afficheur` pourra ressembler à :

```
# fixe un nouveau message a afficher
setMessage(UnTableauDeCaractere)
# un top d'horloge
top()
# renvoie ce qui doit être affiche
to_s()
```

Définir entièrement en Ruby la classe `Afficheur`. Préciser en particulier ses variables d'instances et ce qu'elles représenteront. Écrire en Ruby le code de toutes les méthodes de la classe, sans oublier en particulier les méthodes de création/initialisation. On donne en Annexe 3 un exemple d'utilisation de cette classe `Afficheur`.

C) Les afficheurs avec latence

On remarque que pour les afficheurs de l'exercice précédent il est difficile de voir où se termine le message. Pour éviter ce problème, on veut une nouvelle classe d'afficheurs pour lesquels on pourra spécifier lors de leur création un "*temps de latence*" entre l'entrée du dernier et celle du premier caractère. Ce temps de latence sera exprimé par un nombre positif ou nul d'espaces à insérer entre ces deux caractères.

Appelons `Latence` cette nouvelle classe d'afficheurs.

Définir entièrement en Ruby la classe `Latence`. Préciser en particulier ses variables d'instances et ce qu'elles représenteront. Écrire en Ruby le code de toutes les méthodes de la classe, sans oublier en particulier les méthodes de création/initialisation. On donne en Annexe 4 un exemple d'utilisation de cette classe `Latence`.

D) Les afficheurs avec latence et vitesse paramétrable

A chaque top d'horloge, les afficheurs précédents font un seul décalage. On voudrait une nouvelle sorte d'afficheur dont on pourrait fixer le nombre de décalages effectués à chaque top. Ce nombre sera un entier positif ou nul. Appelons `Vitesse` cette nouvelle classe d'afficheurs

Définir entièrement en Ruby la classe `Vitesse`. Préciser en particulier ses variables d'instances et ce qu'elles représenteront. Écrire en Ruby le code de toutes les méthodes de la classe, sans oublier en particulier les méthodes de création/initialisation. On donne en Annexe 5 un exemple d'utilisation de cette classe `Vitesse`.

Indiquer les relations qui existent entre les différentes classes de cette application. Faire un dessin.

Annexes

Annexe 1 : Rappels de quelques méthodes de la classe `Array` en Ruby

```
[ ] new & * + - << <=> == === [ ] [ ]= | assoc at clear collect compact
concat delete delete_at delete_if each each_index empty? eql? fill first
flatten include? index indexes indices join last length nitems pack pop push
rassoc replace reverse reverse_each rindex shift size slice sort to_a to_ary
to_s uniq Unshift
```

Annexe 2 : Exemple d'utilisation de la classe `Decaleur`

Test	Affichage
`d=Decaleur.creer(5)`	
`puts d`	`<<----->>`
`d.decale('a')`	
`puts d`	`<<----a>>`
`d.decale('b')`	
`puts d`	`<<---ab>>`
`d.decale('c')`	
`puts d`	`<<--abc>>`
`d.decale('d')`	
`puts d`	`<<-abcd>>`
`d.decale('e')`	
`puts d`	`<<abcde>>`
`d.decale('f')`	
`puts d`	`<<bcdef>>`
`puts d.getLargeur`	`5`

Annexe 3 : Exemple d'utilisation de la classe `Afficheur`

Test	Affichage
`leMessage=['D','e','s','p','r','e','s']`	`<<---------D>>`
	`<<--------De>>`
`a=Afficheur.creer(10)`	`<<-------Des>>`
`a.setMessage(leMessage)`	`<<------Desp>>`
	`<<-----Despr>>`
`1.upto(12) do`	`<<----Despre>>`
`  a.top()`	`<<---Despres>>`
`  puts a`	`<<--DespresD>>`
`end`	`<<-DespresDe>>`
	`<<DespresDes>>`
	`<<espresDesp>>`
	`<<spresDespr>>`

Annexe 4 : Exemple d'utilisation de la classe `Latence`

Test	Affichage

`leMessage=['D','e','s','p','r','e','s']`
 	
`a=Latence.creer(10,2)`
`a.setMessage(leMessage)`
 	
`1.upto(12) do`
`  a.top()`
`  puts a`
`end`
	`<<---------D>>`
	`<<--------De>>`
	`<<-------Des>>`
	`<<------Desp>>`
	`<<-----Despr>>`
	`<<----Despre>>`
	`<<---Despres>>`
	`<<--Despres >>`
	`<<-Despres  >>`
	`<<Despres  D>>`
	`<<espres  De>>`
	`<<spres  Des>>`

Annexe 5 : Exemple d'utilisation de la classe `Vitesse`

Test	Affichage

`leMessage=['D','e','s','p','r','e','s']`
 	
`a=Vitesse.creer(10,2,3)`
`a.setMessage(leMessage)`
 	
`1.upto(12) do`
`  a.top()`
`  puts a`
`end`
	`<<------Desp>>`
	`<<--Despres >>`
	`<<spres  Des>>`
	`<<s  Despres>>`
	`<<espres  De>>`
	`<<es  Despre>>`
	`<<Despres  D>>`
	`<<res  Despr>>`
	`<< Despres  >>`
	`<<pres  Desp>>`
	`<<  Despres >>`
	`<<spres  Des>>`