Université du Maine **Licence IFORMATIQUE**

TP N°1 : Introduction à la POO

CONSIGNES pour TOUS les TP de RUBY:

Les TP sont à réaliser dans l'environnement Hop3x.

- Au lancement d'Hop3x vous choisissez la session correspondant au TP.
  - Création d'un projet dont le nom sera TPx
  - Chaque exercice est a réaliser dans un/des fichiers indépendant.
- **Pour chaque exercice**, **vous devez** produire une documentation en utilisant `rdoc`. Pour plus d'information sur l'utilisation de `rdoc` vous pourrez vous rendre sur http://www.ruby-doc.org/stdlib/libdoc/rdoc/rdoc/index.html
  - A partir du travail réalisé avec Hop3x pour générer la `rdoc`. Il suffit de séléctionner « **Générer la rdoc** » depuis le menu « **Langage Ruby** ».
    - En cas de problème, autre solution : lancer la commande `rdoc` depuis un terminal ouvert dans le sous répertoire de Hop3xEtudiant contenant vos sources menu « **Outils/Ouvrir un terminal dans le workspace** »
    - La qualité des commentaires et de la documentation générée prend une part **très importante** dans l'évaluation.
- **Pour toutes les classes** que vous allez créer faire le nécessaire pour pouvoir afficher les objets avec les fonctions d'affichage de Ruby (puts, print, printf, ….)
- En parallèle à l'utilisation d'Hop3x vous pouvez également utiliser `irb` dans un terminal

EXERCICES

1. Normalement c'est déjà fait en partie suite au TD1 :
   - Implémenter les comptes en banques. (programme de test, rdoc).
     Pour les méthodes d'accès **utilisez le coding Assistant**.
   - Implémenter le crible d'Erathosthène. (programme de test, rdoc).

2. Pour la gestion d'une bibliothèque, on nous demande d'écrire une application manipulant des livres. Les livres ont un numéro d'enregistrement, un titre, un auteur, un nombre de pages, un état général ('ok'/'usagé') et un indicateur de disponibilité (true / false)
   1. Définissez en Ruby la classe `Livre`. On souhaite pouvoir créer des livres en fournissant les paramètres nécessaires à l'initialisation des variables d'instances à l'exception de la disponibilité qui devra être initialisée à true et de l'état général qui sera initialisé à 'ok' (bon état). On doit pouvoir consulter les valeurs de toutes les variables d'instance et pouvoir modifier la disponibilité et l'état général.
      - Exemple :
        ```
        liv=Livre.ranger ("Ruby pour les Nul",2,"Jaco",435)
        ```
   2. 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. Modifiez les classes dans ce sens (faire une seconde version)
      - Exemple :
        ```
        liv=Livre.ranger ("Ruby pour les Nul","Jaco",435)
        ```

3. On souhaite pouvoir afficher toutes les informations relatives aux livres, modifier la classe en conséquence.
   - Exemple :

```
print liv
      Livre : Numéro = 2, Titre = Ruby pour les Nul,
      Auteur = Jaco, Nombre de pages=43
```

4. Définissez une classe `Bibliothèque` réduite à une méthode permettant de tester la classe précédente. Les livres créés pourront être stockés dans un tableau (cf classe `Array`). Générez la Rdoc.

---

3. Des maths ! Ajoutez à la classe `Integer` de Ruby les trois méthodes suivantes :
   - `listeDesDiviseurs()` qui retourne dans un tableau la liste des diviseurs de a.
     - Exemple pour a=6 on obtient la liste (1,2,3,6)
   - `listeDesNonsDiviseurs()` qui retourne dans un tableau la liste des non diviseurs de a.
     - Exemple pour a=6 on obtient la liste (4,5)
   - `RapDivNonDiv()` calcule et retourne le rationnel représentant le rapport entre la somme des Diviseurs de a (inférieurs à a) et la somme des nonsDiviseurs de a.
     - pour a= 6 le résultat est 6/9 :
       6 a pour diviseurs inférieurs à 6 : 1, 2, 3 et la somme de ceux-ci vaut 6
       6 à pour non diviseurs inférieurs à 6 : 4 et 5 et la somme de ceux ci vaut 9

On souhaite manipuler ce rapport sous la forme d'un nombre rationnel, donc de la forme `a/b` ou `a/b` est une fraction irréductible (`a/b` est irréductible si elle n'est plus simplifiable). Dans l'exemple précédent on préfèrera donc afficher `2/3` plutôt que `6/9`
Pour cela on vous demande de donner le code de la classe `NR` qui fournit quelques opérations simples sur les nombres rationnels. Les opérations retourneront toujours les résultats sous forme de fractions irréductibles.

La classe `NR` permettra de réaliser les opérations suivantes :

- Construire un rationnel irréductible associé au nombre rationnel a/b.
  - `NR.simplifier(6,9)` permet de créer un rationnel dont le numérateur est 2 et le dénominateur 3
- `ajouter`, `multiplier`, `soustraire` et `diviser` par un rationnel
- afficher un rationnel et déterminer si deux rationnels sont égaux.

Vos classes doivent permettre d'exécuter le programme Ruby suivant :

```ruby
######## Programme de Test ########
puts "\n\tTest de la classe Integer"
puts "\t=========================="
print "Les Diviseurs de 6      = "
p 6.listeDesDiviseurs()
print "Les Nons Diviseurs de 6 = "
p 6.listeDesNonsDiviseurs()
print "Le raport des deux      = "
puts 6.rapDivNonDiv()

puts "\n\tTest de la classe NR"
puts "\t===================="
a=NR.simplifier(6,9)
b=NR.simplifier(5,3)
c=NR.simplifier(15,18)
d= NR.simplifier(5,20)
puts "(#{a}) + (#{b}) = #{e=a.ajouter(b)}"
puts "(#{e}) - (#{c}) = #{f=e.soustraire(c)}"
puts "(#{f}) / (#{d}) = #{g=f.diviser(d)}"
puts "(#{e}) / (#{e}) = #{g.diviser(g)}"
```

```
        Test de la classe Integer
        ==========================
Les Diviseurs de 6      = [1, 2, 3, 6]
Les Nons Diviseurs de 6 = [4, 5]
Le raport des deux      = 2/3

        Test de la classe NR
        ====================
(2/3) + (5/3) = 7/3
(7/3) - (5/6) = 3/2
(3/2) / (1/4) = 6
(7/3) / (7/3) = 1
```

4. **[Exercice FACULTATIF]** Le TDA `Ensemble` en Ruby

   Un ensemble est un type de données permettant de représenter une collection de valeurs de même type. On appelle élément chacune de ces valeurs.

   On rappelle les propriétés principales d'un ensemble :

   - Un ensemble est non ordonné (la place des éléments n'a pas d'importance) et un élément ne peut apparaître qu'une fois.
   - La **suppression** d'un élément n'appartenant pas à l'ensemble est sans effet.
   - Un ensemble peut être vide. La **cardinalité** d'un ensemble est le nombre de ses éléments. Une valeur appartient à un ensemble si elle fait partie de ses éléments.
   - Un ensemble A est **inclus** dans un ensemble B si tous les éléments de A appartiennent à B, deux ensembles A et B sont **égaux** s'ils contiennent les mêmes éléments.
   - La Classe `Ensemble` fournit une méthode permettant d'obtenir l'ensemble union de deux ensembles passés en paramètres. La Classe Ensemble fournit aussi une méthode permettant d'obtenir l'ensemble **intersection** de deux ensembles passés en paramètres.

   Donnez **en Ruby** une réalisation du TDA `Ensemble` en expliquant la structure de donnée concrète retenue.

   1. Ecrire les méthodes de `création/initialisation` d'ensembles
   2. Ecrire les méthodes permettant `l'ajout` et la `suppression` d'élément dans un ensemble, ainsi que le calcul de la `cardinalité`
   3. Ecrire les méthodes de test (`égalité` d'ensemble, `inclusion` d'ensemble, `appartenance` d'un élément, `ensemble vide`)
   4. Ecrire les méthodes `d'union` et `d'intersection` de la classe `Ensemble`
   5. L'ensemble contenant les éléments `A,B,C,D,E` doit s'afficher de la façon suivante : `(A, B, C, D, E)`. Ecrire le code correspondant.
   6. Ecrire un programme de test créant au minimum deux ensembles, et utilisant les méthodes que vous aurez écrites.

   Pour cette exercice on pourra utiliser la classe `Array` dont un extrait de la doc est ci dessous (en résumé on peut utiliser les `Array` comme les tableaux en C).

```
Arrays are ordered, integer-indexed collections of any object.

Array indexing starts at 0, as in C or Java.  A negative index is assumed to be
relative to the end of the array---that is, an index of -1 indicates the last element
of the array, -2 is the next to last element in the array, and so on.

== Creating Arrays
A new array can be created by using the literal constructor [].  Arrays can contain
different types of objects.  For example, the array below contains an Integer, a String
and a Float:
    ary = [1, "two", 3.0] #=> [1, "two", 3.0]
An array can also be created by explicitly calling Array.new with zero, one (the
initial size of the Array) or two arguments (the initial size and a default object).
  ary = Array.new    #=> []
  Array.new(3)       #=> [nil, nil, nil]
  Array.new(3, true) #=> [true, true, true]

== Accessing Elements
Elements in an array can be retrieved using the Array#[] method.  It can take a single
integer argument (a numeric index), a pair of arguments (start and length) or a range.
Negative indices start counting from the end, with -1 being the last element.

 arr = [1, 2, 3, 4, 5, 6]
 arr[2]    #=> 3
 arr[100]  #=> nil
 arr[-3]   #=> 4
```