**Université du Maine** **2012-2013** **Intro POO**

## AUCUN DOCUMENT AUTORISÉ

Il est très fortement recommandé de soigner la présentation de la copie et la qualité des explications fournies, le sujet étant court le correcteur sera impitoyable

Algorithme de correction simplifié : **illisible => Pas lu, pas expliqué => pas compris, A peu près = 0**

## Exercice 1 (7') : Question de cours

1. Si a= "L3 SPI" que représente a[0] ?
2. Quelle erreur commet-on quand on fait hériter la classe `Pile` de la classe `Array` ?
3. Expliquez la différence entre surcharge et redéfinition.
4. Expliquez les différents types de Polymorphisme
5. Expliquer le rôle de l'opérateur <=> en ruby. Donnez en exemple.
6. Expliquez la différence entre une structure de répétition type `while/for` et un itérateur
7. Expliquez le principe de liaison dynamique.

## Exercice 2 (10') : Equations

1. Ecrire en Ruby une classe `Equation`, qui permet de représenter des équations à une inconnue d'un nombre arbitraire de degrés (ex: $8x^4 + 3x^3 + 5x + 12$). **Précisez** la structure de données vous utilisez pour stocker les informations. Ecrire les méthodes nécessaires à la création d'instances d'équation de la façon suivante :
   - `monEquation=Equation.valeur([8 3 0 5 12])` représente l'équation $8x^4 + 3x^3 + 5x + 12$
   - `tonEquation=Equation.valeur([10 0 5 0 0 1])` représente l'équation $10x^5+5x^3+1$
2. Faire le nécessaire pour pouvoir afficher une équation suivant l'exemple donné :
   - `print monEquation -> 3x4 - x2 + 5x + 12`
3. Ecrire les méthodes `plus` et `moins` qui effectuent l'addition et la soustraction de deux équations (chacune renvoie le résultat sous la forme d'une nouvelle équation).

## Exercice 3 (24') : TDA Ensemble

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.
- **L'union** de deux ensembles produit un ensemble contenant les éléments des deux ensembles réunis, alors que **l'intersection** de deux ensembles produit un ensemble contenant les éléments communs aux deux ensembles.
- 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.

- Donnez en Ruby une réalisation du `TDA Ensemble`
  1. Expliquez en Français la structure de données concrête que vous allez utiliser.
  2. Ecrire les méthodes de `création/initialisation` d'ensembles
  3. Ecrire les primitives permettant `l'ajout` et la `suppression` d'élément dans un ensemble,
  4. Ecrire les primitives de test (`égalité` d'ensemble, `inclusion` d'ensemble, `appartenance` d'un élément, ensemble `vide`)
  5. Ecrire les primitives `d'union` et `d'intersection` et de `cardinalité`
  6. Prévoir une primitive d'affichage d'un ensemble
  7. Ecrire un programme de test créant deux ensembles, et utilisant les méthodes que vous aurez écrites.

## Exercice 4 (60') : Code de Cryptage

Le but de cet exercice est de créer et implémenter un modèle orienté objet de codes de cryptage. Important : dans tout ce qui suit, vous considérerez que les chaînes de caractères à crypter sont constituées uniquement de majuscules et ne contiennent pas d'espaces.

### 1) Code de cryptage

Il s'agit ici d'implémenter une classe `Code`, caractérisant des codes de cryptage.

A. Il vous est demandé d'implémenter la classe `Code`, de sorte à ce que les contraintes suivantes soient respectées :
   1. A la création d'un code on doit fournir un `nom` ;
   2. La classe `Code` dispose d'un moyen d'afficher à l'écran le nom du code ;
   3. la méthode de cryptage, `code`, prend en argument le message à crypter (`uneChaine`) et retourne le message crypté (`uneChaine` encore) ;
   4. la méthode de décryptage, `decode`, prend en argument le message à décrypter (`uneChaine`) et retourne le message décrypté (aussi `uneChaine`)

B. Que peut-on dire de la classe `Code` ?

### 2) Trois types de codes de cryptage

Nous considérerons ici trois types de `Code` : les codes à clé secrète, les codes à clé secrète aléatoire et les codes à crans (dits aussi codes de César).

#### 2.1) La classe `ACle` :

Un code à clé utilise une autre chaîne de caractères, nommée clé, pour encrypter/decrypter les messages. Il fonctionne comme suit :

1. à chaque caractère du message en clair, on associe son rang dans l'alphabet : 'A' → 1, 'B' → 2, . . . 'Z' → 26. Pour le message "`COURAGE`" par exemple :

| C | O | U | R | A | G | E |
|---|---|---|---|---|---|---|
| 3 | 15 | 21 | 18 | 1 | 7 | 5 |

2. La clé est aussi transformée en suite d'entiers, selon le même procédé qu'à l'étape précédente. Pour la clé "`MIX`" par exemple :

| M | I | X |
|---|---|---|
| 13 | 9 | 24 |

3. on aligne l'un au dessous de l'autre le message en chiffres et la clé en chiffres (la clé est répétée autant de fois que nécessaire) :

| 3 | 15 | 21 | 18 | 1 | 7 | 5 |
|---|---|---|---|---|---|---|
| 13 | 9 | 24 | 13 | 9 | 24 | 13 |

4. on additionne en colonne les nombres écrits. Lorsque la somme dépasse 26, on diminue le résultat obtenu de 26. Pour l'exemple précédent :

| 16 | 24 | 19 | 5 | 10 | 5 | 18 |
|---|---|---|---|---|---|---|

5. on remplace la suite de nombres obtenue par les caractères correspondants dans l'alphabet :

| P | X | S | E | J | E | R |
|---|---|---|---|---|---|---|

A. Il vous est demandé de programmer la classe en respectant les contraintes suivantes :
   1. A la création de d'un objet on initialise la clé au moyen d'une valeur passée en paramètre ;
   2. la classe doit mettre à disposition un méthode `longueur` retournant la longueur de la clé ;
   3. l'affichage d'un code à clé comprend le nom du code et la clé.

B. Ecrire un programme de test permettant de produire la trace d'exécution suivante :

```
Avec le code : a cle avec EQUINOXE comme cle
Codage de COURAGEFUYONS : HFPAOVCKZPJWG
Decodage de HFPAOVCKZPJWG : COURAGEFUYONS
---------------------------------
```

C. Que peut-on dire de la classe `ACle`

## 2.2) La classe `AcleAleatoire` :

A. Implémentez une classe `ACleAleatoire` permettant de générer aléatoirement une clé de longueur particulière.
   1. A la création d'un objet de la classe on fournit un argument unique (la longueur de la clé). Le nom du code créé est obligatoirement `"a cle aleatoire"`.
   2. Un code à clé Aléatoire dispose d'une méthode sans argument `genereCle` générant aléatoirement une clé de la bonne longueur et l'affectant à l'attribut `clé`

   Indication : la méthode `rand(max=0)` du Kernel Ruby pourra être utilisée pour la génération de nombres aléatoires.
   Ex : `rand(10)` -> un entier compris entre 0 et 9

B. Ecrire un programme de test permettant de produire la trace d'exécution suivante :

```
Avec le code : a cle aleatoire avec XDBFF comme cle
Codage de COURAGEFUYONS : ASWXGEIHAEMRU
Decodage de ASWXGEIHAEMRU : COURAGEFUYONS
 ---------------------------------
```

C. Que peut-on dire de la classe `AcleAleatoire` ?

## 2.3) La sous-classe `Cesar`

Les codes à crans, ou codes de césar, fonctionnent par décalage. Chaque lettre du message est remplacée par la lettre apparaissant n crans plus loin dans l'alphabet (et ce, de façon cyclique : `'Y'` décalé de 4 crans devient `'C'`) Par exemple, le cryptage du message `COURAGE` au moyen d'un code à `3` crans produira : `FRXUDJH`.

La classe `Cesar` est donc caractérisée par le nombre de crans à utiliser pour le cryptage/décryptage. Un code à crans, peut-être vu comme un code à clé où :
   1. la clé est de longueur 1 ;
   2. le caractère occupant la position nombre de crans (modulo 26) dans l'alphabet constitue la clé.

A. Implémentez cette classe en respectant les contraintes suivantes :
   1. A la création d'un objet on fournira le nombre de crans ;
   2. l'affichage d'un code à crans comprend le nom du code et le nombre de crans ;

B. Ecrire un programme de test permettant de produire la trace d'exécution suivante :

```
Avec le code : Cesar a 5 crans
Codage de COURAGEFUYONS : HTZWFLJKZDTSX
Decodage de HTZWFLJKZDTSX : COURAGEFUYONS
-----------------------------------
```

C. Que peut-on dire de la classe `Cesar` ?

3) Donner une représentation graphique des relations entre les différentes classes. Vous préciserez pour chacune les attributs et les méthodes.

## ANNEXE : Documentation de la classe ARRAY

```
Pipounet:~ jacoboni$ ri Array
------------------------------------------------------------ Class: Array
     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.
------------------------------------------------------------------------------
Includes:
---------
     Enumerable(all?, any?, collect, count, cycle, detect, drop,
     drop_while, each_cons, each_slice, each_with_index, entries,
     enum_cons, enum_slice, enum_with_index, find, find_all, find_index,
     first, grep, group_by, include?, inject, inject, map, max, max_by,
     member?, min, min_by, minmax, minmax_by, none?, one?, partition,
     reduce, reject, reverse_each, select, sort, sort_by, take,
     take_while, to_a, to_set, zip)
Class methods:
--------------
     [], new
Instance methods:
-----------------
     &, *, +, -, <<, <=>, ==, [], []=, abbrev, assoc, at, choice, clear,
     collect, collect!, combination, compact, compact!, concat, count,
     cycle, dclone, delete, delete_at, delete_if, drop, drop_while,
     each, each_index, empty?, eql?, fetch, fill, find_index, first,
     flatten, flatten!, frozen?, hash, include?, index, indexes,
     indices, initialize_copy, insert, inspect, join, last, length, map,
     map!, nitems, pack, permutation, pop, pretty_print,
     pretty_print_cycle, product, push, rassoc, reject, reject!,
     replace, reverse, reverse!, reverse_each, rindex, select,
     shelljoin, shift, shuffle, shuffle!, size, slice, slice!, sort,
     sort!, take, take_while, to_a, to_ary, to_s, to_yaml, transpose,
     uniq, uniq!, unshift, values_at, yaml_initialize, zip,
```