# POBJ – Examen Janvier 2013

Durée : 2 heures Tous documents autorisés

---

## PROBLÈME : GÉNÉTIQUE MINIMALISTE

Les processus biologiques qui induisent la création de protéine par décodage du génome sont extrêmement complexes. Lors de cet examen, nous allons créer un modèle ultra-simplifié de ces processus.

Toutes les classes que vous allez créer doivent se situer dans un package *enzyme*.

### 1 : Bases de l'ADN, Codons et acides aminés

Pour commencer, nous allons implémenter plusieurs objets de base du domaine qui nous intéresse.

### 1.1 : Bases

Dans notre modèle simplifié, l'ADN est constitué de 4 bases, qu'on représente par quatre lettres : A, C, G et T.

Q1.1 : Dans le package *enzyme*, créez une énumération qui permet de représenter les 4 bases ci-dessus.

```
package enzyme;

public enum Base {A,C,G,T;}
```

```
Le « ; » est optionnel

-25 % par petite faute : oubli package, majuscule à la place de minuscule, oubli
public... On ne sanctionne ces oublis qu'une seule fois dans la copie, de
préférence sur cette question.
–     50 % pour les erreurs plus sérieuses : « A » au lieu de A, enumeration au
lieu de enum, etc.
–     0 s'ils ne maîtrisent vraiment pas les enums.
```

### 1.2 : Codons

Un codon est une séquence de 3 bases.

Q1.2 : Dans le package *enzyme*, créez une classe *Codon* qui contient un tableau de 3 bases.

Vous doterez cette classe d'un constructeur qui prend en paramètre les 3 bases constituant le codon et d'une méthode *toString()* qui affiche la succession des bases. La classe contiendra aussi les méthodes *equals()* et *hashcode()*. Vous écrirez *equals(), mais* ***hashcode()*** **n'est pas demandée.**

```
package enzyme;

import java.util.Arrays;

public class Codon {
	private Base[] code = new Base[3];

	public Codon(Base a, Base b, Base c){
		code[0]=a;
		code[1]=b;
		code[2]=c;
	}

	public String toString(){
		return ""+code[0]+code[1]+code[2];
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Codon other = (Codon) obj;
		if (!Arrays.equals(code, other.code))
			return false;
		return true;
	}

}
```

```
50 pour equals(), 25 pour constructeur, 25 pour toString()
Je mets les points à equals() si :
-	il prend Object en param,
-	le type est vérifié,
-	il y a un cast en codon
-	on vérifie les bases.
Sinon, 0/50
OK pour utiliser une ArrayList au lieu d'un tableau
-25 pour l'oubli de l'init du tableau
```

## 2 : Des chaînes pour associer ces éléments

A la question 3, nous créerons des chromosomes qui sont des chaînes de codons et des protéines qui sont des chaînes d'acides aminés. Dans cette question, nous nous concentrons sur une implantation générique de ces chaînes, que nous allons réaliser sous la forme de listes chaînées.

Q2.1 : Dans le package *enzyme*, créez une classe *Chainon<T>* qui représente le composant

élémentaire d'une chaîne. Un chaînon connaît la donnée qu'il contient, qui est de type générique T, et son successeur, qui est un autre chaînon. Vous ajouterez toutes les méthodes utiles pour manipuler les éléments de la chaîne ainsi constituée.

```
package enzyme;

/** Classe non public */
class Chainon<T> {

	private T data;
	private Chainon<T> next;

	public Chainon(T data, Chainon<T> next) {
		this.data = data;
		this.next = next;
	}

	public T getData() {
		return data;
	}

	public Chainon<T> getNext() {
		return next;
	}
	public String toString(){
		return data.toString();
	}
}
```

Q2.2 : Dans le package *enzyme*, créez une classe *Chaine<T>*, qui représente une liste chaînée d'éléments génériques et qui implante le comportement *Iterable<T>*. La chaîne connaît principalement le chaînon de tête à partir duquel elle est constituée. Cette classe sera munie d'une méthode *void add(T)* qui permet d'ajouter un élément, d'une méthode *int size()* qui permet de connaître le nombre d'éléments stockés et d'une méthode *iterator()* qui permet d'accéder à son itérateur de type *ChaineIterator<T> (cf. Q2.3)*.

```
package enzyme;

import java.util.Iterator;

public class Chaine<T> implements Iterable<T> {

	private Chainon<T> tete=null;
	private int size=0;

	/**
	 * Ajoute elt en tete.
	 * @param elt
	 */
	public void add (T elt) {
		tete = new Chainon<T>(elt,tete);
		size++;
	}

	/**
	 * Pop elt en tete.
	 * @param elt
```

```
	 */
	public T pop () {
		assert(tete != null);
		T toret = tete.getData();
		tete = tete.getNext();
		size--;
		return toret;
	}

	public int size() {
		return size;
	}

	@Override
	public Iterator<T> iterator() {
		return new ChaineIterator<T>(tete);
	}
	public String toString(){
		String retour  = "[";
		Iterator<T> it = iterator();
		while (it.hasNext()){
			retour += it.next().toString() + ",";
		}
		retour = retour.substring(0, retour.length()-1);
		return retour + "]";
	}

}
```

OK pour ceux qui ont encapsulé une LinkedList (rien ne l'interdit).

pop() non demandé

50 à ceux qui n'ont pas mis de next dans le chainon et ont utilisé une Arraylist pour la chaine

Q2.3 : Dans le package *enzyme*, créez une classe *ChaineIterator<T>* qui permet d'itérer sur une *Chaine<T>*. Cette classe sera munie de toutes les opérations classiques sur les itérateurs, mais on n'implantera pas le *remove()*.

```
package enzyme;

import java.util.Iterator;

class ChaineIterator<T> implements Iterator<T> {

	private Chainon<T> cur;

	public ChaineIterator(Chainon<T> tete) {
		this.cur = tete;
	}

	@Override
	public boolean hasNext() {
		return cur != null;
	}

	@Override
```

```
	public T next() {
		T toret = cur.getData();
		cur = cur.getNext();
		return toret;
	}

	@Override
	public void remove() {
		throw new UnsupportedOperationException();
	}

}
```

remove() non demandé.

Ceux qui ont utilisé LinkedList ou ArrayList ont généralement sauté cette question => 0

Vérifier la cohérence avec les champs de Chainon et Chaine.

## 3 : Table de transcription

Dans notre modèle ultra-simplifié, on suppose qu'il existe une table de transcription simple qui permet de construire un acide aminé à partir d'un codon. Plus précisément, pour chacun des 21 acides aminés possibles, on tire aléatoirement le codon qui lui correspond et on stocke les associations (codon, acide aminé) dans une table associative. Sachant qu'il existe 4*4*4 = 64 codons, de nombreux codons ne correspondent à aucun acide aminé. Tous ces codons dits « non-codants » coderont pour le « stop » (acide aminé de code 0). Par ailleurs, rien n'empêche que deux codons codent pour le même acide aminé.

Q3.1 : Dans le package *enzyme*, créez une classe *TableTranscription* qui permette de réaliser ces associations. Cette classe sera réalisée conformément au pattern *Singleton*.

-50 si rien n'est static, -25 si pas de methode instance()

Q3.2 : Dans la classe *TableTranscription,* ajoutez une méthode *void createTable()* qui réalise une association entre les 21 acides aminés et des codons tirés aléatoirement qui leur correspondent. Pour la construction de codons aléatoires, reportez-vous à la méthode *createRandomCodon()* demandée à la question 4.2.

Il y a un souci dans le corrigé, on peut avoir le même codon plusieurs fois => rejeté dans une HashMap => ignorer cette difficulté.

Q3.3 : Dans la classe *TableTranscription,* ajoutez une méthode *boolean contains(Codon c)*, qui permet de savoir si un codon code pour un acide aminé de code entre 1 et 21 ou non.

Facile => sanctionner (-25%) toute erreur de syntaxe, -50 % pour non maîtrise des HashMaps,

0 si c'est faux

Q3.4 : Dans la classe *TableTranscription,* ajoutez une méthode *AcideAmine get(Codon c)*, qui renvoie l'acide aminé correspondant à un codon.

Facile => sanctionner (-25%) toute erreur de syntaxe, -50 % pour non maîtrise des HashMaps,

0 si c'est faux. Attention à la gestion des codons « non codants » => renvoie AcideAmine(0).

package enzyme;

import java.util.HashMap;

import java.util.Map;

```
public class TableTranscription {

	private static Map<Codon,AcideAmine> table = new HashMap<Codon,AcideAmine>();

	public static void createTable(){
		for(int i=0 ;i<21;i++){
			table.put(ChromosomeFactory.createRandomCodon(), new AcideAmine(i+1));
		}
	}

	public static boolean contains(Codon c){
		AcideAmine am = table.get(c);
		return (am !=null) ;
	}

	public static AcideAmine get(Codon c){
		if(contains(c)){
			return table.get(c);
		}
		return new AcideAmine(0);
	}

	public static String tabletoString(){
		return table.toString();
	}

}
```

## 4 : Chromosomes

Un chromosome est constitué d'une chaîne de codons.

Pour créer des chromosomes, on va réaliser une classe *ChromosomeFactory* qui permet de créer des codons aléatoires et des chromosomes aléatoires constitués d'un nombre aléatoire de codons compris entre 200 et 500.

Q4.1 : Dans le package *enzyme,* créez une classe *Chromosome* qui dispose notamment d'une méthode *toString().*

```
package enzyme;
```

```
public class Chromosome {
	private Chaine<Codon> adn = new Chaine<Codon>();
	private Decodeur decodeur;

	public Chromosome(Decodeur dec){
		decodeur = dec;
	}

	public void add(Codon c){
		adn.add(c);
	}
	public Proteine decode(){
		return decodeur.decode(adn);
	}
}
```

Ne pas sanctionner l'absence de Decodeur à ce stade. Add non demandé.

Q4.2 : Dans le package *enzyme*, créez une classe *ChromosomeFactory*. Dotez-la d'une méthode *Codon createRandomCodon()* qui renvoie un codon aléatoire (toutes les bases sont considérées comme équiprobables).

-25 si pas static

Q4.3 : Dans la classe *ChromosomeFactory*, ajoutez une méthode *Chromosome createRandomChromosome(Decodeur dec)* qui renvoie un chromosome constitué de 200 à 500 codons et lui attribue une méthode de décodage (voir la question 5).

```
package enzyme;

import java.util.Random;

public class ChromosomeFactory {
	private static Random r = new Random();

	public static Chromosome createRandomChromosome(Decodeur dec){

		Chromosome chrom = new Chromosome(dec);
		int longueur = r.nextInt(300)+200;
		for (int i=0;i<longueur;i++){
			chrom.add(createRandomCodon());
		}
		return chrom;
	}

	public static Codon createRandomCodon(){
		Codon codon = new Codon(Base.values()[r.nextInt(4)],Base.values()
[r.nextInt(4)],Base.values()[r.nextInt(4)]);
		return codon;
	}
}
```

-50 si les bornes (200 à 500) ne sont pas bien respectées.
Diverses solutions pour createRandomCodon(), vérifier que c'est correct.

## 5 : Protéines et décodage des chromosomes

Une protéine est constituée d'une chaîne d'acides aminés. On donne ci-dessous le code de la classe *Proteine*.

```
package enzyme;

public class Proteine {
	private Chaine<AcideAmine> composants = new Chaine<AcideAmine>();

	public void add(AcideAmine elt) {
		composants.add(elt);
	}

	public int size() {
		return composants.size();
	}

	public String toString() {
		return composants.toString();
	}

}
```

Pour construire une protéine, un chromosome dispose de diverses méthodes de décodage (représentés par des objets de type *Decodeur*) qui permettent d'engendrer des acides aminés sur la base de ses codons. La protéine résultante est la chaîne d'acides aminés ainsi engendrée.

Le principe de base du décodage est toujours le même : on parcourt tous les codons et, pour chaque codon qui code pour un acide aminé, on ajoute l'acide aminé correspondant à la protéine qu'on est en train de construire.

Cependant, on va réaliser deux façons de décoder. Dans le premier cas (*DecodeurSansStop)*, quand le codon courant code pour un « stop », on l'ignore . On obtient alors une protéine dans laquelle il n'y a pas de « stop ». Dans le second cas (*DecodeurAvecStop)*, on ajoute les « stop » dans la chaîne d'acides aminés constituant la protéine.

Q5.1 : Quel pattern proposez-vous d'utiliser pour représenter les deux façons de décoder les chromosomes en protéines ?

**Strategy. Toute autre réponse = 0**

Q5.2: Ajoutez à la classe *Chromosome* les éléments nécessaires pour la doter d'une méthode *Proteine decode()* qui réalise le comportement décrit ci-dessus. Pour tous les parcours de chaînes, on impose d'avoir recours à un itérateur comme défini à la question 2.3. Vous ajouterez les classes et interfaces, nécessaires pour implanter le pattern identifié à la question 5.1.

```
package enzyme;

public interface Decodeur {
	public Proteine decode(Chaine<Codon> adn);
}
package enzyme;

import java.util.Iterator;
```

```
public class DecodeurAvecStop implements Decodeur {

	public Proteine decode(Chaine<Codon> adn){
		Proteine p = new Proteine();
		Iterator<Codon> iterator = adn.iterator();
		while (iterator.hasNext()){
			Codon c = iterator.next();
			p.add(TableTranscription.get(c));
		}

		return p;
	}
}
package enzyme;

import java.util.Iterator;

public class DecodeurSansStop implements Decodeur{

	public Proteine decode(Chaine<Codon> adn){
		Proteine p = new Proteine();
		Iterator<Codon> iterator = adn.iterator();
		while (iterator.hasNext()){
			Codon c = iterator.next();
			if (TableTranscription.contains(c))
				p.add(TableTranscription.get(c));
		}

		return p;
	}
}
```

Attention à ceux dont la TableTranscription n'est pas static, vérifiez qu'ils sont cohérents

## 6 : Classe principale

Q6 : Créez une classe *ProteineMain* dont la méthode principale *main()* crée un chromosome aléatoire, le décode pour en déduire une protéine, puis affiche la protéine correspondante. Vous donnerez une version décodant sans prendre en compte les « stop » et vous indiquerez ce qu'il faut modifier pour la version qui les prend en compte.

```
package enzyme;

public class ProteineMain {
	public static void main(String args[]){
		TableTranscription.createTable();
		Decodeur dec = new DecodeurAvecStop();
		Chromosome c = ChromosomeFactory.createRandomChromosome(dec);
		Proteine p = c.decode();
		System.out.println(p);
	}
}
```

-50 pour mauvaise gestion des éléments static