TD 1 - Introduction au Système Unix et à Java

1. Quelques commandes de base sous GNU/linux

a) Chemin, répertoire

Taper `pwd` Où est-on ? Ce répertoire par défaut est dénommé « home »

Taper `ls` puis `ls -l`

Rôle de `mkdir` (voir le `man`)

Créer un répertoire `test` à l'intérieur du répertoire « home »

Voir ce que contient ce répertoire avec `ls test`.

Se déplacer dans le répertoire `test` à l'aide de `cd test`

Vérifier le répertoire courant.

b) Éditeur de texte

Lancer `gedit`

Si les numéros de lignes ne sont pas affichés, les faire apparaître (Édition → Préférences → Affichage).

Régler de même le paramètre de retour à la ligne : décocher « Activer ».

Créer un fichier, y inscrire son nom, le sauver sous le nom `truc.txt`

Voir les fonctions de base de l'éditeur : curseur, effacement arrière / avant, copier / couper / coller, sauver / sauver sous...

Quitter `gedit`.

Taper `cd ..` Où est-on ? Taper à nouveau `cd ..`

Regagner le répertoire home avec `cd` sans argument.

c) Copie de fichier

Créer le répertoire `mib_xyz` en remplaçant `xyz` par un nom au choix.

Utiliser `cp` pour copier le fichier `truc.txt` de `test` dans `mib_xyz`.

Aller dans `mib_xyz` ; Vérifier le contenu avec `ls`, puis le contenu du fichier avec `cat truc.txt`.

Supprimer `truc.txt`

Nous allons maintenant écrire notre premier programme avec `gedit` (voir l'utilisation de `gedit &`).

2. Java et les classes

a) Définitions

Les **classes** sont des modèles qui se réalisent concrètement dans des objets. Pour chaque classe, on peut créer et utiliser un ou plusieurs **objets** : les **instances** de la classe.

Une classe contient des **données membres** (ou **attributs**) et des **fonctions membres** (ou **méthodes**).

Les méthodes contiennent des instructions, définissant les actions effectuées sur les attributs.

On créera toujours une classe par fichier : classe `Bonjour` dans `Bonjour.java` (par convention, les noms de nos classes commencent par des majuscules, mais ce n'est pas obligatoire).

b) Programme `Milieu.java`

Exemple 1.1

Construisons un premier exemple de programme Java, destiné à calculer la charge nette d'une protéine. Pour cela, nous créons :

- une classe `Proteine`, destinée à représenter la protéine. Pour l'instant, notre classe ne nécessite que trois attributs : le nombre d'AA chargés positivement, le nombre d'AA chargés négativement, et la charge nette à pH=7, qui se déduit des deux premiers attributs. Un méthode permet d'effectuer le calcul.
- une classe `Milieu`, qui correspond au programme proprement dit, lequel va utiliser un (ou plusieurs) objet(s) de la classe `Proteine`.

Les trois données membres de la classe `Proteine` sont des nombres. Le langage Java permet de définir de tels nombres grâce à des **types prédéfinis** (ou **primitifs**). Les nombres entiers sont de type `int`, les nombres à virgule peuvent être de type `double`. Nous déclarons donc trois attributs de type `int` : `nbPos`, `nbNeg`, `charge`.

La fonction membre de `Proteine` est nommée `calcCharge()`. La parenthèse est impérative (même si, comme ici, elle est vide) car c'est elle qui indique qu'il s'agit d'une fonction et non d'une donnée. Si la fonction nécessitait des paramètres, ceux-ci seraient indiqués dans la parenthèse (nous verrons comment plus tard). Le corps de la fonction est écrit entre accolades. Ici, il s'agit d'une seule instruction, qui calcule la valeur d'une donnée en fonction de l'autre. La fonction effectue ce calcul, range le résultat dans une variable, mais ne retourne aucun résultat, comme l'indique le terme `void` (vide) qui la précède.

Voici une représentation de la classe `Proteine`[1] :

Proteine

`int nbPos`
`int nbNeg`
`int charge`

`void CalcCharge()`

La classe `Milieu` permet d'exploiter la classe `Proteine`. Pour être exécutable, elle doit contenir une fonction `main()` (principale). Ici `main()` est la seule méthode. Nous avons donc :

1 Cette représentation s'inspire de la grammaire UML. Pour une introduction (en anglais) à UML, voir par exemple le tutoriel `http://www.objectmentor.com/resources/articles/umlClassDiagrams.pdf` et les références citées.

Milieu
`void main()`

L'en-tête de la méthode `main()` est une ligne assez compliquée dont nous détaillerons les éléments plus tard. Pour l'instant, en voici une description simplifiée :

- `public` permet d'utiliser une méthode ou un attribut depuis une autre classe.
- `static` implique que la classe n'est instanciée que dans un seul objet, une seule fois, lors de l'exécution du programme. Le programme est nécessairement `static`, il n'y a qu'un `main()`, une seule fonction principale, un seul **point d'entrée** dans le programme.
- `void` signifie que `main()` ne renvoie pas d'information en retour.
- `String args[]` signifie que `main()` peut recevoir éventuellement des informations, au travers de variables arguments, en entrée (nous utiliserons cette possibilité plus tard).

La fonction `main()` commence par créer `prot1`, une instance de la classe `Proteine`, puis elle donne une valeur à ses membres `nbPos` et `Neg`. Elle appelle ensuite la fonction `calcCharge()` de l'objet `prot1`, pour lui faire calculer et stocker dans le membre `charge` la charge nette à pH=7 de la molécule. Enfin, elle affiche la valeur de l'attribut `charge` de `prot1` dans un message lisible à l'écran.

Pour cette dernière action, la fonction `main()` de la classe `Milieu` fait appel à une méthode définie dans une classe que nous n'avons pas écrite nous-mêmes :

- `System` est une classe pré-existante[2], appartenant à la bibliothèque de base de Java. Son attribut `out` (instance de la classe `PrintStream`) comporte diverses méthodes utiles pour l'affichage.
- `println()` est une méthode de `System.out`, on lui passe comme argument la chaîne de caractères `"Charge = "`, suivie du membre `charge` de l'objet `prot1`, puis `" à pH=7."`. Les chaînes de caractères entre guillemets sont affichées *in extenso*, tandis que les données membres sont interprétées : c'est leur valeur qui est affichée.

Enfin, notons l'utilisation de commentaires, qui sont destinés à l'usage des lecteurs humains du code :

- `/* ... */` Commentaire, éventuellement sur plusieurs lignes ;
- `//` Commentaire sur la ligne courante (jusqu'à la fin).

c) Compilation du code Java

Le programme que nous venons d'écrire est lisible et compréhensible par un humain (qui connaît le langage Java) mais tel quel, il ne peut pas être exécuté par une machine. Il nous faut auparavant traduire notre code en instructions directement exécutables. Naturellement, nous n'allons pas effectuer cette traduction nous-mêmes : c'est un programme, le **compilateur**, qui va s'en charger. Le compilateur spécifique du langage Java s'appelle `javac` (pour « java compiler »).

L'instruction `javac Milieu.java` génère autant de fichiers `.class` que de classes utilisées par le programme. Ici, `Proteine.class` et `Milieu.class`.

---

2 Pour une decription exhaustive (en anglais) des classes prédéfinies du langage Java standard, on peut consulter par exemple : `https://docs.oracle.com/javase/7/docs/api/overview-summary.html`

Les fichiers `.class` sont écrits en **bytecode** (littéralement : code en octets). Un tel fichier se compose donc d'un ensemble d'octets, qui correspondent à des instructions qu'il nous serait difficile de comprendre, mais qui peuvent être exécutées.

d) Exécution d'une application

Une application Java comporte au minimum une classe qui contient une méthode `main()`. **Exécuter** une application consiste à appeler sa méthode `main()`, qui est son point d'entrée. L'exécution n'est pas un processus autonome : elle nécessite un environnement capable d'exécuter le bytecode. Ce peut être un processeur Java, qui interprète et exécute directement le bytecode. Mais le plus souvent, un tel processeur est en fait simulé par une **machine virtuelle**. La machine virtuelle est une couche logicielle interprétant (ligne à ligne) le bytecode et le traduisant en instructions compréhensibles par la machine cible (définie par son processeur et son système d'exploitation) sur laquelle on travaille. Le même bytecode peut ainsi être exécuté sur différentes plateformes (PC sous MS-windows ou GNU/linux, Ordinateur MacIntosh sous MacOSX, PDA sous PalmOS, etc.). Nous pouvons utiliser, comme machine virtuelle, un navigateur web ou un programme dénommé « `java` ». Nous utilisons `java`, que nous lançons en lui donnant le nom (sans extension) de la classe à exécuter (qui doit donc contenir `main()`) :

```
java Milieu
```

Exercice 1.1

Enrichir l'exécutable `Milieu` pour qu'il traite deux instances de la classe `Proteine`, (au lieu d'une seule comme jusqu'à présent).

Exercice 1.2

Enrichir la classe `Proteine`. en y ajoutant deux attributs : ses deux coordonnées `x` et `y` sur une grille plane (par convention, l'origine est en haut à gauche). Une méthode `deplaceADroite()` permettra de déplacer la protéine d'une unité vers la droite.

Modifier la classe Milieu, afin qu'elle positionne les deux protéines, les déplace à droite, puis affiche leurs coordonnées à l'écran.

Ajouter et utiliser de même des méthodes pour déplacer une protéine à gauche, en haut, en bas.

Exercice 1.3

Remplacer les instructions d'affichage dans les fonctions `main()` de `Milieu`, par l'appel à de nouvelles méthodes que l'on créera dans la classe `Proteine`: `afficheCharge()` et `affichePosition()`.

COMPRENDRE

- Le fonctionnement de base du système d'exploitation, de l'éditeur de texte.
- La notion de classe en tant que modèle d'un objet réel.
- Les classes et leurs instances, les membres et l'accès avec la notation pointée.
- La compilation du source et l'exécution du *bytecode*.

TD 2 – Encapsulation - Graphisme avec AWT

1. Encapsulation

a) Nécessité de l'encapsulation

Dans les programmes que nous avons écrits jusqu'ici, l'utilisation des membres d'une classe est réalisée grâce à la notation pointée : nous avons écrit des expressions telles que `prot1.nbNeg` ou `prot2.charge`. Il s'agit donc d'un accès direct à la structure interne de la classe, ce qui ne va pas sans poser quelques problèmes.

Exemple 2.1

Il est possible de créer des objets absurdes, comme une molécule ayant un nombre négatif d'AA d'un type (positif ou négatif). De plus, rien ne nous protège contre les incohérences : nous pourrions créer une protéine ayant une charge nette qui ne corresponde pas à son nombre de charges positives et négatives : il suffit d'affecter une valeur à `charge`, sans utiliser la méthode `calcCharge()`, ou de modifier `nbPos` et/ou `nbNeg` en omettant d'appeler `calcCharge()`. Ce type de problème peut sembler mineur si l'on considère que la même personne écrit la classe qui modélise l'objet, et la classe exécutable. Mais ce n'est pas toujours le cas, et l'un des objectifs d'une bonne programmation est d'obtenir un code facilement réutilisable, soit par une autre personne, soit par soi-même (quand on reprend son propre code six mois plus tard, on est bel et bien une autre personne).

De plus, si nous souhaitons que les objets informatiques soient des représentations fidèles des objets réels, il est naturel d'inclure, dans la constitution-même d'une classe, les relations entre ses attributs et ce qu'il est possible de faire faire à cette classe. Réutilisation du code et pertinence de la modélisation sont les raisons majeures de l'introduction du concept d'**encapsulation** dans les langages « à objets ».

b) Principe d'encapsulation

L'idée d'encapsulation est que *les données membres d'une classe ne doivent être manipulées directement qu'à l'intérieur de la classe à laquelle elles appartiennent*, ce qui assure modularité et sécurité d'utilisation.

Exemple 2.2

Pour empêcher l'accès direct au membre `charge` de la classe `Proteine`, nous devons faire précéder sa déclaration du mot `private`. Il est alors interdit d'écrire `prot1.charge` dans `Milieu`. ou toute autre classe qui utilise des objets de la classe `Proteine`.

L'accès aux attributs `private` depuis « l'extérieur » se fera exclusivement à travers une (ou plusieurs) méthode(s) de la classe `Proteine`. Par exemple, pour afficher la valeur de `charge`, on ne peut plus écrire `System.out.println(prot1.charge)` dans `Milieu`. Il faut utiliser la méthode `afficheCharge()` que nous avons écrite précédemment. Cette méthode doit évidemment être déclarée publique (`public`) pour que `Milieu` puisse l'appeler. En fait, on pourrait omettre ce terme car méthodes et attributs sont `public` par défaut. Cependant, écrire explicitement l'accessibilité permet au programmeur de clarifier ses intentions.

c) Accesseurs

Un programme qui utilise la classe `Proteine` peut aussi avoir besoin de connaître la charge, et non de l'afficher. Nous allons donc écrire une méthode `getCharge()`, qui permet de récupérer la valeur du

membre `charge`. Cette méthode produit un résultat, et son type est donc celui du résultat, ici `int`. Le contenu de cette méthode est simplement `return charge;`.

Il nous faut aussi écrire des méthodes (publiques) pour modifier les autres données membres, par exemple `setPos()` pour donner une valeur à `nbPos`. Une telle méthode doit recevoir un argument : la valeur à attribuer à la donnée. Pour ce faire, il faut déclarer que cette méthode doit être appelée avec un paramètre, dont on spécifie le type : `setPos(int val)`. Le corps de la méthode attribue simplement à `nbPos` la valeur reçue : `nbPos=val`. Une telle méthode, qui a pour rôle de donner une valeur à un attribut ou de connaître la valeur d'un attribut, s'appelle **accesseur**.

En revanche, la charge n'est jamais attribuée directement, puisqu'elle est une conséquence des valeurs de deux autres membres. Nous n'écrivons donc pas de méthode telle que `setCharge(int val)`.

Notre nouvelle classe `Proteine` se présente désormais ainsi :

Proteine

`private int nbPos`
`private int nbNeg`
`private int charge`
`private int x,y`

`public void setPos(int val)`
`public int getPos()`
`public void setNeg(int val)`
`public int getNeg()`
`public void CalcCharge()`
`public int getCharge()`
`public void afficheCharge()`
`public void setX(int abs)`
`public int getX()`
`public void setY(int ord)`
`public int getY()`
`public void deplaceADroite()`
`public void deplaceAGauche()`
`public void deplaceEnBas()`
`public void deplaceEnHaut()`
`public void AffichePosition()`

Nous devons maintenant remplacer dans `Milieu`, les accès directs aux attributs par des appels aux méthodes. Ainsi, `prot1.nbPos=-4;` s'écrit désormais `prot1.setPos(-4);`.

Exercice 2.1

Compléter `Proteine` et `Milieu` pour encapsuler aussi les membres `nbNeg`, `x` et `y`.

d) Encapsulation, cohérence, redondance

Exemple 2.3

Notre première version ainsi encapsulée représente un progrès, mais il reste possible de faire faire des choses absurdes à une `Proteine`. En particulier, nous pourrions oublier de calculer la charge après avoir modifié `nbPos` ou `nbNeg`. La cohérence de notre objet `Proteine` impose au contraire que la charge se déduise des valeurs de `nbPos` et `nbNeg`. Il est donc inutile de disposer d'un attribut `charge` : cette donnée peut être systématiquement calculée « à la volée », par les méthodes qui en ont besoin, ici `afficheCharge()` et `getCharge()`. La méthode `calcCharge()` disparaît donc égalementde `Proteine`, et il est évidemment désormais impossible de l'appeler depuis `Milieu`.

Il est important de chercher à préserver la cohérence à l'intérieur d'un objet, et d'éviter les redondances. C'est ici une même transformation de l'objet qui permet de satisfaire ces deux exigences.

Attention : les méthodes `afficheCharge()` et `getCharge()` peuvent être appelées avant même qu'une valeur ait été affectée à `nbPos` ou `nbNeg`. En conséquence, le calcul risque d'avoir lieu lieu avant que les attributs aient une valeur (évidemment, ce n'est pas une conséquence de l'encapsulation, ni de la suppression de l'attribut `charge` : on pouvait aussi appeler `calcCharge()` hors de propos dans la version précédente). Certains compilateurs et/ou machines Java n'acceptent pas qu'on utilise une variable avant de lui avoir donné une valeur. D'autres donnent par défaut une valeur aux variables (0 pour les variables numériques). Pour éviter tout risque d'erreur, nous donnons explicitement par défaut la valeur 0 à `nbPos` et `nbNeg`.

Voici l'objet `Proteine` après cette transformation :

Proteine

`private int nbPos`
`private int nbNeg`
`private int x,y`

`public void setPos(int val)`
`public int getPos()`
`public void setNeg(int val)`
`public int getNeg()`
`public int getCharge()`
`public void afficheCharge()`
`public void setX(int abs)`
`public int getX()`
`public void setY(int ord)`
`public int getY()`
`public void deplaceADroite()`
`public void deplaceAGauche()`
`public void deplaceEnBas()`
`public void deplaceEnHaut()`
`public void AffichePosition()`

2. Graphisme avec AWT

a) introduction

L'environnement Java offre un certain nombre des classes prédéfinies, notamment pour construire des applications graphiques. Un élément graphique important est représenté par la classe `Frame`, incluse dans le paquetage `awt` (Abstract Window Toolkit - utilitaire de fenêtre). On peut donc associer à une application une **fenêtre**, en créant un objet `Frame`. La classe contient un certain nombre de fonctions, dont trois sont indispensables ici :

- la fonction `setVisible(boolean b)` qui affiche ou fait disparaître la fenêtre selon la valeur de `b` (`true` ou `false`),
- la fonction `setSize(int width, int height)` qui permet de donner à la fenêtre une largeur de `width` pixels et une hauteur de `height` pixels.
- la fonction `getGraphics()` qui renvoie une référence à un objet de la classe `Graphics`. C'est cet objet qui constitue le « contexte graphique » de l'affichage, sur lequel vont concrètement s'effectuer les dessins. Une fenêtre (objet de la classe `Frame`) doit être visible pour disposer d'un contexte graphique.

Donc, si l'on appelle `getGraphics()` avant `setVisible(true)`, la valeur renvoyée par `getGraphics()` sera `null`.

Attention ! Une fenêtre (`Frame`) n'est pas un programme. C'est un simple objet de visualisation, qui va permettre de dessiner. Elle est associée à certaines fonctions de gestion prédéfinies, mais en nombre restreint. Par exemple, il est possible de redimensionner la fenêtre, mais les boutons situés en haut à droite ne sont pas tous fonctionnels : l'action sur le bouton « fermeture » n'est pas prédéfinie. Il faudrait donc rattacher à la fenêtre une fonction spécifique traitant l'appui sur le bouton « fermeture ». Pour limiter la complexité de notre programme, nous nous passerons de cette fonctionnalité. La fenêtre devra donc être fermée en terminant l'application qui l'a créée depuis le terminal (par `[Ctrl]-C`).

Autre limitation : une fenêtre doit être munie d'une fonction de ré-affichage complète. En effet, en cas de modification de la taille de la fenêtre, ce n'est pas le système qui la redessine. Il ne fait qu'appeler une fonction `paint()` associée à la fenêtre. Il faudrait réécrire cette fonction `paint()` pour toute fenêtre permettant le redimensionnement. Ceci ne sera pas fait dans l'immédiat. Donc, si on affiche un dessin dans la fenêtre, ce dessin disparaîtra si on modifie la taille de la fenêtre.

b) Application

La simulation graphique permet de visualiser des phénomènes, ou du moins le modèle qu'on se donne pour les représenter.

Exemple 2.4

Dans un premier temps, le programme crée trois objets, deux objets `Proteine` et un objet `Frame` (qui reçoit, lors de sa création, un titre à afficher dans le bandeau de la fenêtre). Disposant d'une fenêtre, il est possible de dessiner la protéine des exercices précédents, sous forme d'un disque coloré. Pour ce faire, nous dotons la classe `Proteine` d'une méthode `dessine()`. Les instructions de dessin fonctionnent toutes de la même manière :

- on sélectionne une couleur grâce à la méthode `setColor()`. L'argument de `setColor()` est une couleur, qu'on peut indiquer soit en utilisant une couleur prédéfinie (ce sont des attributs de la classe `Color`, nommés `white`, `black`, `red`, `blue`, `pink`, etc.), soit en créant une couleur avec `new Color(r,g,b)` où `r`, `g` et `b` sont des nombres de 0 à 255 représentant respectivement les intensités des composantes rouge, verte et bleue de la couleur.
- on dessine en utilisant une des méthodes de la classe `Graphics` : `drawOval()`, `fillOval()`, `fillRect()`, etc. Le dessin utilise la dernière couleur sélectionnée. Attention donc à ne pas oublier de changer de couleur si nécessaire !

Notre méthode `dessine()` utilise l'instruction `fillOval()` pour dessiner une élipse (dont le cercle est un cas particulier) colorée. Ses arguments sont, dans l'ordre : l'abscisse du point en haut à gauche, l'ordonnée de ce même point, la largeur, la hauteur. Comme `setColor()` et `fillOval()` sont des méthodes de la classe `Graphics`, il nous faut transmettre une instance de `Graphics` à la méthode `dessine()`. Notre code ne contient cependant pas d'instruction du type `Graphics g = new Graphics()`, car le contexte graphique est en fait créé par l'objet `Frame` que nous avons instancié dans `Milieu` : nous récupérons une référence à cet objet grâce à `getGraphics()`, et nous transmettons cette référence à `dessine()` lors de chaque appel.

Exercice 2.2

Faire afficher de même la protéine `prot2`, puis faire visualiser les mouvements obtenus à l'aide des méthodes de déplacement de la classe `Proteine`. Afin de rendre les déplacements plus visibles, ils s'effectueront désormais par pas de 10.

Exercice 2.3

Modifier la méthode `dessine()` pour que la couleur d'une protéine dépende de sa charge nette. La couleur pourra être un dégradé allant du bleu (très positive) au rouge (très négative). On considère que les protéines que nous manipulons ont une charge comprise entre -12 et +12.

COMPRENDRE

- L'encapsulation, la différence entre membres `public`/`private`.
- L'utilisation des classes et paquetages prédéfinis, des outils graphiques.

TD3 – Tests

1. Sélection

L'encapsulation et l'utilisation d'accesseurs nous a permis d'améliorer la cohérence de notre objet `Proteine`. Cependant, il reste des risques d'incohérence. Par exemple, dans `Milieu.java`, il reste possible d'écrire `prot1.setPos(-126);` bien que cela n'ait aucun sens. Évidemment, on peut imaginer que l'utilisateur de l'objet `Proteine` soit prévenu de ce risque et fasse attention. Mais il est plus sûr de considérer que l'objet `Proteine` soit lui-même responsable des limites imposées à ses données membres. Pour cela, il faut **tester**, à l'intérieur de la méthode `setPos()`, si la valeur imposée de l'extérieur est acceptable ou non.

On utilise pour cela une structure de **sélection** :

```
if (condition)
     {action(s) 1}
else
     {action(s) 2}
```

Le fonctionnement d'une sélection est le suivant :

1. La *condition* est évaluée ;
2. Si le résultat de 1. est vrai (`true`), alors le bloc d'instructions {*action(s) 1*} est exécuté, puis on saute directement au point 4 ;
3. Si le résultat de 1. est faux (`false`), alors le bloc {*action(s) 2*} est exécuté (s'il y en a un : `else...` est optionnel), puis on saute au point 4 ;
4. Le programme se poursuit avec les instructions qui suivent.

NB Dans l'écriture de la sélection, les accolades ne sont pas obligatoires s'il n'y a qu'une action.

Exemple 3.1

Si le nombre de charges positives est négatif, le programme affiche un message d'erreur et met ce nombre à 0 ; sinon il affecte à `NbPos` la valeur du paramètre.

Exercice 3.1

Modifier de même la méthode `setNeg()` ;

Modifier les méthodes de déplacement afin qu'une protéine qui sort du cadre par un côté y rentre par le côté opposé.

Exercice 3.2

Ajouter une donnée membre de type caractère (`char`) nommée `config` qui représentera l'état de la protéine : phosphorylée (`P`) ou non phosphorylée (`N`). On écrira pour l'instant une méthose `setConfig()` qui ne vérifiera pas la validité du caractère. La méthode `afficheConfig()` devra afficher « `protéine phosphorylée` » si `config` vaut `'P'`, « `protéine non phosphorylée` » sinon.

Modifier la méthode `dessine()` pour que l'intensité de composante verte dépende de `config`.

Écrire une méthode `changeConfig()` qui inverse la configuration de la protéine.

2. Opérateurs booléens

Dans un test, on peut avoir besoin de spécifier une condition qui est une combinaison de plusieurs conditions. Pour ce faire nous disposons des **opérateurs booléens** (ou **relationnels**) classiques ET, OU et NON, qui s'écrivent respectivement en Java : `&&`, `||` et `!`.

Exemple 3.2

La méthode `setX()` peut imposer que la valeur donnée à `x` soit comprise entre 0 et 599. Si ce n'est pas le cas, elle attribue à `x` la valeur 300 :

```
public void setX(int abs) {
  if ( ( abs > 0 ) && ( abs < 600 ) )
    x = abs;
  else {
    System.out.println("Valeur x="+abs+"impossible => x=300");
    x = 300;
  }
}
```

Exercice 3.3

Modifier de même la méthode `setY()`.

Modifier `setConfig()` pour que `config` ne puisse pas prendre d'autre valeur que '`N`' ou '`P`'.

COMPRENDRE

- Les expressions booléennes et leur utilisation pour l'écriture de la structure de sélection.

TD 4 - Constructeurs et agrégation

1. Constructeur simple

L'identificateur (le nom) d'un objet ne représente qu'une **référence** à l'objet effectif. La valeur de cette référence correspond à l'adresse de l'objet. Ce dernier n'existe (et donc ne peut être utilisé) qu'après avoir été créé par l'appel de l'opérateur `new`. Tout objet n'est accessible qu'à travers une référence, mais plusieurs références peuvent désigner le même objet (elles contiennent en fait la même adresse). Inversement, le fait d'appeler l'opérateur `new` plusieurs fois sur la même référence lui fera désigner successivement des objets différents, et chaque nouvel appel rendra l'objet précédent inaccessible par cette référence.

Pour créer un objet (réserver l'espace mémoire nécessaire à son stockage), l'opérateur `new` appelle en fait une méthode particulière de la classe : son **constructeur**. Toute classe se voit automatiquement attribuer un constructeur, sans que nous ayons à l'écrire explicitement. Mais il est souvent nécessaire d'écrire soi-même un constructeur pour une classe, de façon à faire un peu plus de choses que la simple attribution de mémoire à une instance. En particulier, on peut, au moyen d'un constructeur, **initialiser** les attributs d'un objet à des valeurs « par défaut », ou prendre garde à assurer une certaine cohérence entre les différentes données membres.

Exemple 4.1

Pour écrire un constructeur qui positionne automatiquement une nouvelle protéine au centre de la grille 600x600, nous créons la méthode `Proteine()` :

```
public Proteine() {
  x = 300;
  y = 300;
}
```

ou mieux :

```
public Proteine() {
  setX(300);
  setY(300);
}
```

Cette méthode porte nécessairement le nom de la classe, et n'a pas de type de retour (pas même `void`). Elle doit évidemment être `public`.

Exercice 4.1

Modifier le constructeur de la classe `Proteine`, afin qu'il initialise la configuration à l'état "non phosphorylé", et les nombres de charges positives et négatives à 0.

2. Constructeur avec paramètre(s)

Les constructeurs que nous venons de voir fabriquent un objet « par défaut », que nous devrons ensuite modifier à l'aide des méthodes d'accès aux attributs. Il est également possible d'initialiser un objet avec des

valeurs choisies par le programme qui crée l'objet. Pour ce faire, nous devons écrire un autre constructeur, qui prenne en compte un ou plusieurs **paramètre(s)**.

Exemple 4.2

Un constructeur pour initialiser les nombres de charges positives et négatives de la classe `Proteine` (noter l'utilisation de `setPos()` et `setNeg()`) :

```
public Proteine(int np, int ng) {
  setX(300);
  setY(300);
  setConfig('N');
  setPos(np);
  setNeg(ng);
}
```

Dans `Mileu.java`, on doit passer les arguments (dans l'ordre !) au constructeur lorsqu'on crée un objet de la classe `Proteine`.

Les deux constructeurs peuvent coexister dans la même classe. L'un ou l'autre sera appelé en fonction de l'instruction qui crée l'objet.

Exercice 4.2

Modifier le constructeur avec paramètres de la classe `Proteine`, afin qu'il initialise au moyen de paramètres la position et la configuration.

3. L'agrégation (composition)

D'une manière générale, toute la logique de la programmation actuelle repose sur l'élaboration de code/structures complexes à partir de « briques » simples. En pratique, un code n'est donc jamais écrit totalement à partir de rien, mais en utilisant des éléments existants. Cette complexification n'est possible que si les fragments de code, ou les objets élémentaires sont correctement structurés. En particulier, dans le cas des objets, si les règles d'encapsulation ont été respectées.

Un objet est une variable structurée, obtenue par **agrégation** de plusieurs informations (données membres) et méthodes. Les données membres peuvent être de type simple prédéfini (primitives), ou être elles-mêmes des objets. On a entre ces objets une relation d'agrégation, permettant de construire un objet complexe par réunion d'objets plus simples.

L'agrégation permet d'utiliser les propriétés élémentaires des constituants pour aboutir à des propriétés globales de l'objet structuré. Inversement, on répercute sur les objets de base des actions globales sur l'objet structuré.

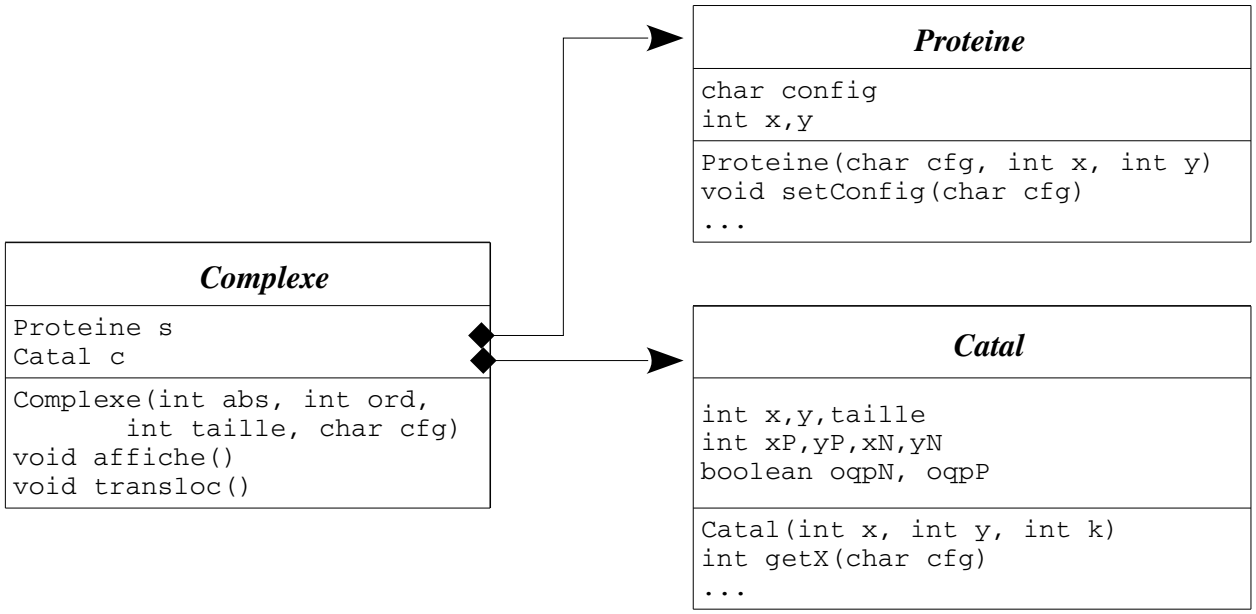
Exemple 4.3

Un complexe enzyme-substrat peut être formé par **composition** de deux classes. Le substrat est simplement une `Proteine`. La nouvelle classe `Catal` est imitée de la classe `Proteine`. On y trouve des membres x et y, et un nouveau paramètre de forme : la taille (on assimilera la forme du catalyseur dans le plan à un carré).

On supposera qu'il s'agit d'une phosphorylase à action réversible, qui comporte deux sites : un site P pour la forme phosphorylée du substrat et un site N pour la forme non phosphorylée. Chacun de ces sites se trouve au centre d'un bord de l'enzyme (site N sur le bord haut, site P sur le bord gauche) et sa position est donnée par les méthodes `getXsite(char conf)` et `getYsite(char conf)`. Enfin, un booléen

décrit l'état de chacun des sites : libre ou occupé. La méthode `fixe(char conf)` permet de fixer un substrat sur un des deux sites, selon sa configuration.

Le complexe contient une enzyme (`Catal`) et un substrat (`Proteine`). Cette relation de composition sera représentée ainsi :



Le constructeur de `Complexe` doit créer et initialiser une instance de `Proteine` et une instance de `Catal`. La classe exécutable `Milieu` déclare un `Complexe`, et n'accède jamais directement aux instances de `Catal` ou `Proteine`.

Exercice 4.3

Écrire dans la classe `Complexe` une méthode `transloc()`, qui change la configuration le substrat, ce qui s'accompagne d'un changement de site de la protéine. Mettre en évidence la phosphorylation et la translocation au cours de l'exécution.

COMPRENDRE

- La notion de constructeur.
- la réalisation de constructeurs avec ou sans paramètres.
- L'agrégation, qui consiste à construire un objet d'une certaine complexité à partir d'objets plus simples.
- Les relations entre les objets élémentaires et l'objet structuré.

TD5 - Boucles, tableaux et chaînes de caractères

1. Boucle `while`

La boucle constitue le moyen de répéter des instructions plusieurs fois. Il existe en Java (et dans d'autres langages) deux types de boucles communément utilisés. La première consiste à répéter une instruction tant qu'une certaine condition est vérifiée. C'est la boucle `while` :

```
while(condition)
  { action(s) }
```

L'écriture ressemble à celle du test, mais si la condition est remplie, après avoir effectué l'action (ou les actions) de l'intérieur de la boucle, le flot d'exécution au lieu de continuer, remonte à l'instruction `while` et recommence.

1. On évalue *`condition`*. Si le résultat est vrai (`true`), on saute au point 2, sinon, on saute au point 4 ;
2. Les *`action(s)`* sont exécutées ;
3. On retourne au point 1 ;
4. On poursuit avec les instructions suivantes du programme.

Exemple 5.1

On peut modifier la méthode `setX()` de la classe `Proteine` de façon à ce que si l'abscisse donnée en paramètre est supérieure à 599, elle soit ramenée dans l'intervalle [0,599] en lui soustrayant 600. Mais rien ne garantit qu'une soustraction de 600 suffise, car l'abscisse peut être supérieure à 1199, et même 1799, etc. Une boucle s'impose.

Exercice 5.1

Procéder de même pour les abscisses inférieures à 0. Reporter cette modification dans la méthode `setY()` et dans les méthodes équivalentes de la classe `Catal`.

Exercice 5.2

Écrire une nouvelle classe exécutable `Reaction` (inspirée de `Milieu`) dans laquelle une protéine est positionnée au hasard, puis se déplace aléatoirement en changeant chacune de ses coordonnées d'au maximum 5 unités. Le déplacement aléatoire se poursuivra jusqu'à ce que la distance de la protéine au centre (300,300) soit inférieure ou égale à 40.

Pour tirer un nombre aléatoire, il faut déclarer et initialiser une instance de la classe `Random` (ce qui nécessite d'importer `java.util.*`). Cette classe contient la méthode `nextInt(int max)` qui renvoie un nombre aléatoire[3] dans l'intervalle [0,max[. Afin de pouvoir réutiliser le déplacement aléatoire, nous l'implémenterons sous forme d'une méthode de la classe `Proteine` : `bougeAlea(int nb)` fait se déplacer la protéine dans les deux directions d'au maximum `nb` pas.

La racine carrée se calcule avec la méthode `sqrt(int val)` de la classe `Math`, qui est une classe

3 Il n'existe pas de vrai hasard en informatique. Les nombres renvoyés par `nextInt()` sont en fait pseudo-aléatoires : ce sont les éléments d'une suite générée par une fonction ré-entrante (on calcule $x_n = f(x_{n-1})$). Cette fonction doit être assez chaotique pour que la suite satisfasse un certain nombre de critères statistiques (moyenne, distribution des écarts entre deux valeurs successives, etc).

statique (donc pour laquelle on ne créera pas d'instance) disponible par défaut (donc utilisable sans importer de bibliothèque). La méthode `Math.sqrt(int val)` renvoie un nombre de type `double`.

2. Boucle `for`

La boucle `for` est en principe utilisée lorsque le nombre de répétitions est déterminé avant l'entrée dans la boucle. Sa forme générale est la suivante :

```
for(initialisation;condition;instruction finale)
  { action(s) }
```

L'exécution d'une boucle `for` se déroule ainsi :

1. On effectue l'*initialisation*. ;
2. On évalue *condition*. Si le résultat est vrai (`true`), on saute au point 3, sinon, on saute au point 6 ;
3. Les *action(s)* sont exécutées ;
4. On effectue l'*instruction finale* ;
5. On retourne au point 2 ;
6. On poursuit avec les instructions suivantes du programme.

L'utilisation la plus commune de la boucle `for` consiste à lui faire compter le nombre de répétitions. Les trois éléments de la parenthèse concernent alors une variable appelée *compteur*. Par exemple :

```
for ( i=0 ; i<5 ; i++)
  System.out.println("Bonjour numéro "+i);
```

Dans cet exemple, `i` sert de compteur : il est initialisé à 0, puis à chaque tour comparé à 5 puis (après que le message ait été affiché) incrémenté. L'instruction de la boucle sera donc exécutée 5 fois. Noter qu'il est possible (et même fréquent) de déclarer le compteur dans la parenthèse du `for`, auquel cas il n'est bien sûr accessible que dans la boucle :

```
for ( int i=0 ; i<5 ; i++)
  System.out.println("Bonjour numéro "+i);
```

Exemple 5.2

Dans la classe `Milieu`, on peut faire exécuter dix translocations successives au complexe.

Exercice 5.3

Remplacer la boucle `while` de `Reaction` par une boucle `for` qui fait toujours effectuer 100 pas aléatoires à la protéine.

3. Tableaux

Une forme très utilisée de construction d'informations structurées à partir de données simples est réalisée par les **tableaux**. Un tableau est un regroupement d'éléments de même type, éléments qui sont repérés par un **index**, de type entier et commençant toujours à 0.

Le type des éléments peut être un type simple (`int`, `double`, `char`, `boolean`, etc.) ou une classe prédéfinie ou créée par nous-mêmes.

a) Tableau de type simple

Comme tout objet en Java, un tableau n'est accessible qu'à travers une référence. Ils n'existe qu'après avoir été créé par `new` (et il disparaît s'ils n'est plus référencé). La syntaxe pour créer un tableau est :

```
type identifiant [] = new type(taille);
```

Exemple 5.3

Il est possible de construire un tableau pouvant contenir 100 nombres réels (type `double`) et le remplir avec les valeurs successives de la distance de la protéine au centre. Afin d'harmoniser le nombre de pas et la taille du tableau, on utilisera une variable et/ou l'attribut `length` du tableau.

Exercice 5.4

Stocker de même les valeurs de `x` et de `y` dans deux nouveaux tableaux. Après la fin des déplacements de la protéine, afficher la liste des valeurs successives prises par x, y et la distance au centre.

b) Tableau d'objets

Un tableau d'objets est créé de la même façon qu'un tableau de type simple, mais une fois le tableau créé, chacune de ses cases contient la valeur `null`. Il faut alors créer les éléments du tableau un par un, en utilisant à chaque fois une instruction `new`.

Exemple 5.4

Dans `Milieu`, nous pouvons manipuler plusieurs instances de `Complexe`, stockées dans un tableau. Après création du tableau, chaque case est instanciée avec un nouveau `Complexe`. Nous pouvons déclencher la translocation de ces éléments l'un après l'autre.

c) Chaînes de caractères

Dans une certaine mesure, une chaîne de caractères peut être vue comme un tableau de caractères. Il est ainsi possible de déclarer :

```
char nom [] = new char(6);
```

On peut ensuite entrer des caractères un à un dans ce tableau :

```
nom[0] = 'Z' ;
nom[1] = 'o' ;
nom[2] = 'r' ;
nom[3] = 'r' ;
nom[4] = 'o' ;
nom[5] = '!' ;
```

Tout ceci devient vite fastidieux, mais Java dispose d'une classe spécifique pour les chaînes de caractères : la classe `String`. Une instance de la classe `String` s'initialise en une seule opération :

```
String nom = new String("Zorro!");
```

Il y a d'autres différences entre un tableau de caractères et une instance de `String`. Celles qui nous importent pour l'instant sont :

- La longueur d'une `String` est obtenue par la méthode `length()`;
- L'accès à un caractère se fait à l'aide de la méthode `charAt(int p)`, qui renvoie le p^{e} caractère de la chaîne.

Exemple 5.5

Disposant des tableaux d'objets et du type `String`, nous pouvons créer une nouvelle version de la classe `Proteine`, en y intégrant la séquence en acides aminés :

- Un attribut de la protéine est sa séquence[4]. Cette séquence est représentée par un tableau d'acides aminés que nous nommons `sequence`.

4 Attention ! Ce n'est pas le seul : une protéine ne se réduit pas à sa séquence. Par exemple, l'état de phosphorylation n'est pas indiqué par la séquence, c'est un paramètre distinct.

- Nous créons une classe `AcAm` destinée à représenter un cide aminé. Dans cette classe, le caractère représentatif d'un AA (son symbole) est une des caractéristiques d'un résidu AA. L'autre caractéristique dont nous avons besoin est la charge à pH=7. Nous nous en tiendrons à ces attributs, mais une représentation plus complète pourrait intégrer, selon les besoins, une masse, un encombrement, une formule chimique, etc. Les méthodes membres de la classe AcAm sont simples : un constructeur et deux accesseurs.

AcAm
`char symbole` `int chph7`

`AcAm(char s)`
`char getSymb()`
`int getChpH7()`

- De la séquence pourront se déduire le nombre de charges positives et le nombre de charges négatives à pH=7. Les attributs correspondants peuvent donc disparaître *sans changer l'interface de la classe `Proteine`* : les méthodes `getPos()` et `getNeg()` existent toujours, seul change leur code. En revanche, les méthodes `setPos()` et `setNeg()` sont désormais inutiles (ce n'est pas grave : nous n'appelions pas ces méthodes depuis l'extérieur, seul le constructeur les utilisait).
- Nous supprimons le constructeur sans paramètres : il lui faudrait une séquence par défaut, dont on voit mal comment justifier l'existence.
- Le constructeur avec paramètres de la classe `Proteine` change : il n'est plus question d'indiquer directement les nombres de charges positives et négatives en tant que paramètres. A la place, il faut lui donner une chaîne de caractères (type `String`) qui représente la séquence de la protéine.

Pour faire fonctionner cette nouvelle version de `Proteine`, nous créons une protéine en passant sa séquence lors de la construction.

COMPRENDRE

- Les boucles de type `while` et `for`.
- Les tableaux : le mécanisme d'indexation et la déclaration.
- Les tableaux de types simples, d'objets, et la classe `String`.

TD 6 - Algorithmique

1. Introduction

Un **algorithme** est la description d'une suite d'opérations qui servent à exécuter une tâche donnée. C'est une sorte de « programme en langage naturel », c'est-à-dire qu'il ne respecte pas une syntaxe aussi stricte qu'un code source. Il doit cependant constituer une formulation parfaitement explicite et structurée, susceptible d'être traduite (implémentée) dans un langage quelconque.

Les ingrédients d'un algorithme sont les *structures de contrôle* que nous avons vues lors des séances précédentes :

- *Séquence* d'instructions, exécutées dans l'ordre dans lequel elles sont écrites
- *Sélection* d'instructions qui peuvent être exécutées ou non, selon le résultat d'un test
- *Répétition* d'instructions

Toute procédure de calcul peut en définitive s'exprimer selon un agencement de ces structures. C'est en cela que consiste le travail d'algorithmique. Pour obtenir un algorithme à partir d'un problème un peu complexe, il est nécessaire de recourir à l'*analyse descendante*, c'est-à-dire à une décomposition du problème en sous-problèmes, qui sont eux-mêmes décomposés, etc. jusqu'à obtenir des micro-problèmes faciles à résoudre.

2. Exemple : recherche de motif

a) Analyse

Nous allons écrire un programme capable de trouver, dans une séquence protéique, toutes les occurrences d'un motif (courte séquence). C'est, par exemple, ce que ferait un programme qui cherche à identifier des sites de phosphorylation. Afin de simplifier le programme, nous considérons pour l'instant qu'un site de phosphorylation est caractérisé par le motif `ILHVGF`. Notre programme doit donc trouver sur une protéine toutes les positions où ce motif apparaît.

Pour écrire ce programme, nous allons d'abord procéder à une analyse descendante. Au passage, nous établirons une représentation des données dont nous avons besoin. Dans notre cas, une première décomposition pourrait être :

```
Rechercher un motif mot dans une séquence pro =
    Créer la séquence pro et le motif mot
    Afficher les positions auxquelles le motif mot apparaît dans la séquence pro
```

Pour pouvoir continuer, il est indispensable que la séquence de *mot* soit plus courte que la séquence de *pro* : nous devons le vérifier, et stopper le programme si ce n'est pas le cas. La longueur d'une protéine est celle de son tableau `sequence` (cf TD 5) mais pour simplifier l'écriture de l'algorithme, nous appellerons les longueurs de *pro* et *mot*, respectivement lp et lm. Nous pouvons donc considérer que ce sous-problème est complètement décomposé :

```
Créer la séquence pro et le motif mot =
    Créer la séquence pro et lire son contenu ; mettre sa longueur dans lp
    Créer la séquence mot et lire son contenu ; mettre sa longueur dans lm
    Si lm > lp
```

```
    afficher un message « erreur... » et arrêter
  Sinon
    continuer
```

Le terme « continuer » désigne la suite de l'algorithme, c'est-à-dire le second sous-problème, pour lequel une nouvelle décomposition s'impose.

```
Afficher toutes les positions auxquelles le motif mot apparaît dans la séquence pro =
  Parcourir la séquence pro, et à chaque position p :
    Si mot apparaît à partir du p-ième AA de pro
      afficher p
```

De toute évidence, « Parcourir la séquence » consiste à faire une boucle, dans laquelle p (un entier) va prendre toutes les valeurs possibles. Quelles sont-elles ? Sur la séquence, le premier résidu amino-acide est numéroté 0, et le dernier lp-1. Nous convenons que p désigne la position de l'AA de la séquence qui correspond au début du mot. La valeur minimum de p est donc 0. La dernière position où l'on peut trouver le motif dépend bien sûr de lp, mais également de lm, par exemple :

```
    0                                                   lp-1
    |                                                   |
pro DEQSAVACIGGLRKTINVALLDDLKVGDYVILHVGFALQKLDEAEAQRT
mot ILHVGF
    |    |
    0    lm-1


                                                    lp-lm
    0                                               |    lp-1
    |                                               |    |
pro DEQSAVACIGGLRKTINVALLDDLKVGDYVILHVGFALQKLDEAEAQRT
mot                                                 ILHVGF
                                                    |    |
                                                    0    lm-1
```

Ainsi, notre boucle peut s'écrire plus précisément :

```
    Pour p variant de 0 à lp-lm :
    ...
```

Il nous faut maintenant détailler la comparaison entre *mot* et la partie de *pro* qui commence en position p. Cette comparaison consiste simplement à vérifier successivement l'égalité des AA de *pro* et de *mot* qui sont en correspondance. Par exemple :

```
    0               p    p+lm-1                         lp-1
    |               |    |                              |
pro DEQSAVACIGGLRKTINVALLDDLKVGDYVILHVGFALQKLDEAEAQRT
mot                 ILHVGF
                    |    |
                    0    lm-1
```

On vérifie si l'AA de *pro* en position p est égal au nucléotide de mot en position 0, ce que nous noterons *pro*[p]==*mot*[0]. Ensuite, il faut comparer *pro*[p+1] avec *mot*[1], etc. jusqu'à la fin du motif, c'est-à-dire *pro*[p+lm-1]==*mot*[lm-1]. Il s'agit donc d'une nouvelle boucle, dont l'indice i varie de 0 à lm-1, et dans laquelle on vérifie à chaque tour la valeur de vérité de l'expression booléenne

pro[*p*+i]==*mot*[*i*]. Pour que le mot soit trouvé dans la séquence en position *p*, il faut que tous les AA soient identiques, c'est-à-dire que toutes les expressions booléennes renvoient *true*. En d'autres termes, un ET logique appliqué à toutes ces expressions doit avoir la valeur *true*. Comme les expressions de comparaison sont évaluées successivement dans la boucle, on ne peut pas écrire directement l'expression :

trouvé = ((*pro*[*p*]==*mot*[0]) && (*pro*[*p*+1]==*mot*[1]) ...)

Le plus simple est alors de combiner les résultats des comparaisons successives l'un après l'autre, et de garder le résultat dans la variable booléenne trouvé. Nous utilisons ainsi la propriété d'associativité :

(a ET b ET c) = ((a ET b) ET c)

Dans notre cas, on aura à chaque tour à effectuer l'opération booléenne :

trouvé = *trouvé* ET (*pro*[*p*+*i*]==*mot*[*i*])

La variable trouvé doit évidemment être initialisée avec la valeur *true* (pour s'en convaincre, il suffit d'imaginer ce qui se passe si on l'initialise à *false* !) à chaque nouvelle position examinée.

Notre algorithme complet ressemble finalement à ceci :

```
Rechercher un motif mot dans une séquence pro =
   Créer les données =
      Créer la séquence pro et lire son contenu ; mettre sa longueur dans lp
      Créer la séquence mot et lire son contenu ; mettre sa longueur dans lm
   Si lm > lp :
      afficher un message d'erreur et arrêter
   Sinon :
      Afficher toutes les positions auxquelles mot apparaît dans pro =
         Pour p variant de 0 à lp-lm :
            trouvé = true
            Pour i variant de 0 à lm-1
               trouvé = trouvé ET (pro[p+i]==mot[i])
            Si trouvé
               afficher p
```

b) Programmation

Nous implémentons notre algorithme de recherche de mot dans `Milieu`. La création des données nécessite, dans le programme principal, une simple déclaration et initialisation d'une séquence protéique : `pro` et d'un motif : `mot`. Ces deux séquences jouent un rôle différent dans le programme. Pour `pro`, nous pouvons utiliser la classe `Proteine` et pour `mot`, une simple chaîne de caractère suffira.

La phase de création de `pro` peut être réalisée grâce au constructeur qui prend en paramètre une chaîne de caractère (`String`) représentant la séquence. En conséquence, `pro` et `mot` seront codés « en dur » dans le source[5].

Les longueurs des séquences, que dans l'algorithme nous avons notées *lp* et *lm*, seront accessibles différemment pour les deux objets. Nous avons la méthode `length()` pour accéder à la longueur de `mot`. Pour `pro`, il nous faut accéder à la longueur du membre `sequence` depuis l'extérieur de l'objet `Proteine`, et pour cela nous créons une méthode `getLen()`.

Les tests et boucles s'implémentent naturellement avec des `if` et `for`.

5 Nous verrons plus tard comment donner le contrôle à l'utilisateur, en prenant en compte les paramètres donnés au programme sur la ligne de commande, ou en lisant les données dans un fichier.

Dans l'algorithme, l'accès à un AA particulier d'une séquence est écrit sous la forme $pro[i]$. Dans notre classe `Proteine`, la séquence est en fait un attribut, codé sous forme d'un tableau d'instances de la classe `AcAm`. Son i-ème nucléotide correspond à `pro.sequence[i]`. Mais bien sûr, encapsulation oblige, nous n'accéderons pas ainsi directement au membre `sequence` d'une protéine, ni bien sûr à son contenu. Il nous faut doter la classe `Proteine` d'une nouvelle méthode qui réalise cet accès. Comme `pro.sequence[i]` représente une instance de la classe `AcAm`, nous ne pouvons pas le comparer directement à un caractère de `mot`. Notre méthode d'accès à un AA de la protéine doit renvoyer le membre `symbole` du i-ème élément du tableau `sequence`.

Exemple 6.1

On peut modifier `Milieu` de façon à ce que les 10 translocations aient lieu si et seulement si la protéine comporte le motif de phosphorylation. Comme la protéine est contenue dans un `Complexe`, il faut que ce dernier dispose d'un constructeur adéquat, et permette d'accéder aux informations concernant la séquence de son membre `sub`.

c) Une petite amélioration

On peut noter que lors de la comparaison entre la séquence et le motif, il est en général inutile de comparer tous les AA : on peut s'arrêter dès qu'un AA différent est rencontré. On peut donc remplacer la boucle `for` par une boucle `while` qui continue tant qu'aucune différence n'est constatée, et que la fin du motif n'est pas atteinte.

Exercice 6.1

Modifier en conséquence l'algorithme, et implémenter de plus cette nouvelle version sous forme d'une méthode de la classe `Proteine` qui écrit sur la console chaque occurrence du motif et renvoie `true` si au moins une occurrence a été trouvée, `false` sinon. Utiliser cette nouvelle méthode dans `Milieu`.

Exercice 6.2

On dispose d'un ensemble de molécules (stockées dans un tableau). On souhaite les classer en fonction de leur taille, c'est-à-dire avoir la molécule de plus petite taille dans la première case du tableau, puis les molécules de taille croissante dans les cases successives, jusqu'à la plus grande molécule dans la dernière case. Établir l'algorithme et écrire le programme correspondant.

On pourra implémenter cet algorithme dans une classe `Tri`, inspirée de `Milieu`, en y ajoutant les méthodes nécessaires (comme pour la méthode `delay()` ces méthodes seront définies avant la méthode `main()`) et en travaillant sur des instances (une douzaine) de `Catal` de taille aléatoire (par exemple entre 5 et 20), les dessiner et/ou afficher leur taille dans l'ordre du tableau, puis trier le tableau et dessiner et/ou afficher son étét final. Il nous faudra donc utiliser un tableau d'objets de type `Catal`.

COMPRENDRE

- Notions d'analyse descendante et d'algorithme.
- Implémentation : passage de l'algorithme au programme.

TD 7 - Interactions entre objets

1. Introduction

Dans le modèle « objet », chacun des objets est considéré comme une entité autonome, avec son propre contenu et ses propres méthodes pour interagir avec l'extérieur. Mais les objets peuvent ainsi agir directement avec d'autres objets.

Prenons l'exemple simple de deux objets : une proteine et un catalyseur. La protéine se déplace, et lorsqu'elle arrive dans la zone de fixation du catalyseur, ce dernier la fixe et la modifie. Ceci peut être géré de deux façons :

- Programmation procédurale classique : toutes les relations sont traitées dans le programme principal. C'est ce dernier qui contient la boucle dans laquelle la protéine se déplace, puis à chaque déplacement teste si la protéine est arrivée dans la zone de fixation du catalyseur.
- Programmation par messages (ou événements) : la boucle de déplacement de la protéine est traitée dans une méthode de la classe `Proteine`. A chaque pas, la protéine envoie une information au catalyseur, lui indiquant qu'elle s'est déplacée. Le catalyseur traite alors cette information, et détermine s'il doit fixer ou non la protéine, selon sa position. Dans cette approche, le programme principal n'intervient plus du tout au cours de la boucle : son rôle se borne à créer les deux objets, établir initialement leurs relations, puis démarrer la boucle de déplacement de la protéine en appelant la méthode correspondante.

L'intérêt de cette deuxième approche repose sur le découplage des objets : leurs relations ne sont plus envisagées globalement, mais individuellement. Chaque objet connaît « au départ » les autres objets avec lesquels il doit interagir. Ceci permet de concevoir des programmes de grande taille, dans lesquels de nombreux objets interagissent entre eux selon des modes d'actions en chaîne, dont toutes les configuration de séquence d'actions ne peuvent pas être analysées de manière exhaustive. C'est en particulier le cas des programmes offrant une interactivité avec l'utilisateur : la nature et l'ordre des actions ne peuvent plus être complètement prévus à la conception du programme. Chaque action agira donc spécifiquement sur tel ou tel objet, qui lui-même pourra répercuter l'action sur d'autres objets.

2. `Observable` et `Observer`

A partir de l'exemple proteine / catalyseur, on peut définir les spécifications de la programmation :

- la protéine se déplace et signale son déplacement au catalyseur. Elle doit donc avoir accès à ce dernier. Concrètement, un objet de la classe `Proteine` doit posséder une référence vers une instance de `Catal` ;
- le catalyseur devra recevoir l'information du déplacement, et y répondre par une action spécifique.

On pourrait envisager d'utiliser simplement un attribut « pointeur vers le catalyseur » dans la protéine, attribut initialisé au départ par le programme principal. Mais ce problème d'interaction étant général, deux classes prédéfinies existent pour le traiter en Java : `Observable` et `Observer`.

- la classe `Observable` possède les deux méthodes nécessaires pour la protéine :
    - `addObserver()` pour enregistrer tous les objets auxquels on doit envoyer les messages ;

    - `notifyObserver()` pour envoyer effectivement les messages aux objets concernés, plus une méthode `setChanged()` nécessaire pour gérer le lien entre changement d'état et envoi du message.
- l'**interface** `Observer`. Ce n'est pas réellement une classe, mais la définition de l'interface que doit implémenter tout objet jouant ce rôle d'**écouteur**. Ces objets doivent obligatoirement déclarer une méthode : `update()`, qui est la méthode de traitement spécifique.

Il faut noter la différence conceptuelle entre les deux représentations :

- dans la classe `Observable`, les méthodes existent déjà : `addObserver()` enregistre dans une liste les objets auxquels on enverra les messages, et `notifyObserver()` balaye cette liste pour envoyer effectivement les messages. Toutefois, cette action ne peut se faire qu'en appelant une méthode commune à tous les récepteurs, méthode qui doit donc être définie ailleurs (nom, liste de paramètres).
- l'interface `Observer` définit effectivement la méthode `update()`. Mais le contenu de cette méthode est différent selon les objets. Elle ne peut donc pas être construite en dehors du contexte spécifique du programme. Elle doit toutefois être connue pour que la classe `Observable` puisse l'appeler. Cette méthode n'est donc déclarée que par son interface, qui devra être ensuite complété par la définition réelle du code : c'est le rôle de la structure « interface » utilisée en Java.

Exercice 7.1

Faire de la classe `Proteine` une **classe dérivée** de `Observable`, et de la classe `Catal` une **implémentation** de l'interface `Observer`. Le code correspondant à ce programme reprendra les classes `Proteine` et `Catal` déjà utilisées, avec les modifications suivantes :

- dans la classe `Proteine`, le dessin est fait en dehors du contexte du programme principal. Le `Graphics` utilisé doit donc être mémorisé comme attribut par le constructeur. Une méthode `efface()` est ajoutée. L'attente par une méthode `delay()` est aussi intégré dans la classe. La méthode `vivre()` représente la boucle d'action de la `Proteine`. L'attribut `libre` et les méthodes associées permettent de représenter l'état libre/fixé de la protéine.
- dans la classe `Catal`, le `Graphics` est aussi intégré. La méthode `update()` est implémentée. Noter que son paramètre étant de type de base `Observable`, il doit être **transtypé** en `Proteine` pour accéder aux méthodes spécifiques de cette classe.

3. Objets graphiques

Pour répondre aux besoins de la construction de programmes munis d'interfaces graphiques, l'environnement Java propose de nombreux objets graphiques prédéfinis. Les actions associées à ce objets sont donc prédéfinies aussi, et leur mode de communication spécifié.

Exemple 7.1

La classe `Button` représente un objet graphique de type bouton. Outre ses propriétés graphiques (taille, intitulé,...), cet objet est capable d'envoyer des messages selon la logique définie plus haut : il enregistre la liste des objets auxquels il doit envoyer l'information « bouton pressé ». Ce sont ses « écouteurs ». La classe `Button` possède donc une méthode `addActionListener()`.

Le type d'action associé peut être défini de différentes façons. On utilise ici un nom, mis en place par la méthode `setActionCommand()`. La méthode `actionPerformed()`, implémentée dans la classe `Proteine`, permet de tenir compte de l'action pour modifier la valeur d'un attribut.

COMPRENDRE

- Principe de la programmation par interaction entre objets.
- Notion d'interface.

TD 8 - Héritage I

1. Introduction

L'**héritage** est un mécanisme permettant de **spécialiser** une nouvelle classe à partir d'une classe existant déjà (la classe **de base**). On parle également de **dérivation**.

Cette spécialisation peut prendre plusieurs formes :

- enrichissement de la classe de base par création de nouvelles méthodes ou de nouvelles données,
- modification de méthodes existant déjà, pour les remplacer par de nouvelles méthodes spécifiques : c'est le **polymorphisme**.

Conceptuellement, l'héritage peut s'appliquer dans des approches graduées :

- dans le cas le plus simple, il ne fait que spécialiser la classe de base. Cette classe existe donc, des objets peuvent être instanciés à partir de la classe. Mais certains objets ont des propriétés spécifiques, ce qui justifie la construction d'une classe dérivée.
- dans la forme la plus aboutie, il traduit directement une hiérarchie conceptuelle dans laquelle tous les objets sont au même niveau hiérarchique, et appartiennent eux-mêmes tous à des classes dérivées d'une classe de base unique. A la limite, on peut envisager que les objets effectifs ne puissent appartenir qu'à une des classes dérivées, mais jamais à la classe de base, qui ne sert alors que de modèle de départ. Cette classe devient alors une classe abstraite. Ceci sera l'objet du prochain TD.

Nous avons déjà utilisé l'héritage au TD précédent, quand nous avons fait dériver `Proteine` de la classe `Observable`. Il s'agissait alors d'utiliser les propriétés définies dans une classe pré-existante.

Dans l'analyse d'un problème donné, il est également fréquent de constater que la notion d'héritage s'impose au cours du développement, en amenant la généralisation de plusieurs classes, initialement conçues séparément (éventuellement successivement), mais présentant suffisamment de points communs pour justifier le regroupement.

Il importe de bien distinguer agrégation et héritage : l'agrégation correspond à l'expression « est formé de », alors que l'héritage traduit la relation « est un » : un complexe est formé d'un catalyseur et une protéine (agrégation), alors que le catalyseur est une molécule (héritage).

2. Héritage

a) Regroupement des données communes

Si deux (ou plusieurs) classes possèdent des attributs et/ou des méthodes en commun, alors ces classes peuvent être considérées comme des spécialisations d'une même classe ancêtre.

Exemple 8.1

Les classes `Proteine` et `Catal` possèdent en commun le fait d'être positionnées dans une fenêtre au moyen de coordonnées pouvant prendre des valeurs comprises entre 0 et 599. Cela se traduit par la présence dans ces deux classes de :

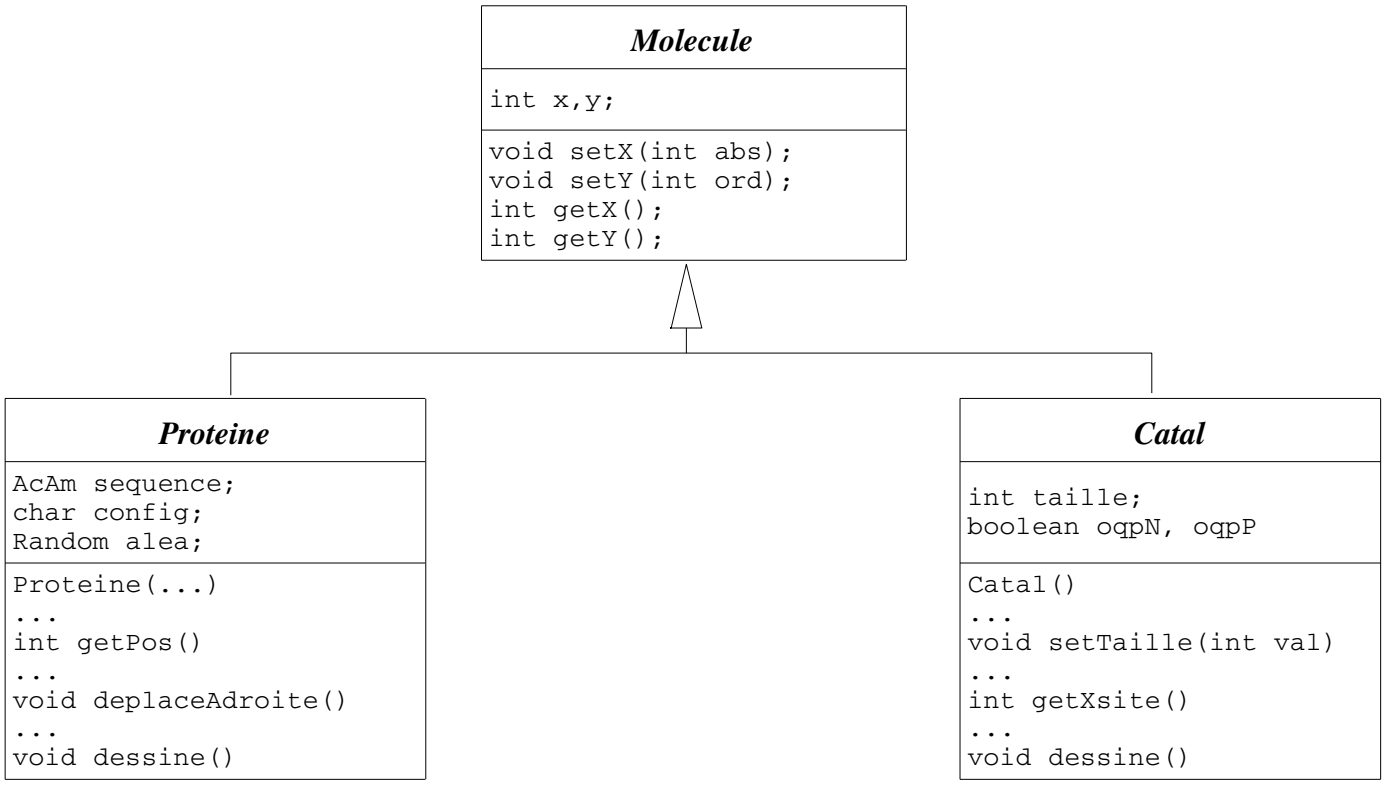
- deux attributs numériques, `x` et `y` pour représenter la position de l'objet ;

- des méthodes pour positionner l'objet et connaître sa position.

On peut faire « remonter » ces attributs et méthodes dans une classe de base : `Molecule`.

Les autres méthodes restent spécifiques, ainsi que `dessine()`, que nous traiterons plus tard.

Nous représenterons la relation d'héritage de la façon suivante :



Noter que les classes dérivées ne peuvent pas accéder aux données déclarées `private` dans la classe de base. Si les classes dérivées doivent utiliser ces données, il faut en changer les conditions d'accès en les déclarant `protected` au lieu de `private`. C'est un assouplissement du mécanisme d'encapsulation, mais limité aux classes dérivées.

Enfin, rappelons que la dernière version de `Proteine` que nous avons écrite est déjà dérivée : elle hérite de la classe `Observable`. Or, une classe ne peut être dérivée que d'une seule classe ancêtre. Mais heureusement, l'héritage est transmis entre niveaux successifs. Donc pour que `Proteine` devienne descendant de `Molecule`, tout en restant descendant d'`Observable`, nous faisons de `Molecule` un descendant d'`Observable`. Ainsi, toute classe dérivée de `Molecule` est aussi dérivée d'`Observable`.

Exercice 8.1

Écrire :

- une classe `MolMob` pour représenter les molécules mobiles. Cette classe doit dériver de `Molecule`, et être ancêtre de `Proteine`.
- une classe `Phosphate` pour représenter les ions Phosphate susceptibles de se fixer sur une protéine. Cette classe `Phosphate` est naturellement un autre descendant de la classe `MolMob`.

Pour tester ces classes, créer une instance de `Phosphate` dans `Observation`. Afin que les deux molécules mobiles bougent simultanément, modifier la méthode `vivre()` de sorte qu'elle n'agisse que durant un pas de temps, et faire effectuer la boucle par `Observation`.

NB Les directives d'importation de paquetages (awt, util...) sont locales aux fichiers sources des classes. Elles n'ont pas d'effet dans les classes dérivées et doivent donc être répétées, si besoin est, dans les fichiers des classes dérivées.

b) Héritage et constructeurs

Dans l'état actuel de notre hiérarchie de classes, chaque classe sans descendant possède un constructeur (ou plusieurs), alors que les classes de niveaux supérieurs n'en ont pas. Ces constructeurs effectuent des opérations spécifiques de leur classe, mais aussi des opérations communes à deux ou plusieurs classes. On peut améliorer notre modèle en créant des constructeurs au niveau des classes ancêtres.

Comme toute classe, une classe dérivée possède un constructeur par défaut qui est appelé au moment de l'initialisation d'une instance. Ce constructeur par défaut appelle en fait le constructeur par défaut de la classe ancêtre. Si l'on doit définir un constructeur pour la classe dérivée, alors l'appel au constructeur de l'ancêtre est nécessairement sa première instruction. En l'absence d'appel explicite, c'est le constructeur par défaut de l'ancêtre qui est appelé. Mais on peut aussi faire un appel à un constructeur "surchargé" de la classe ancêtre. Cet appel remplace alors l'appel implicite au constructeur par défaut. Il permet d'utiliser un constructeur avec paramètres, avec la syntaxe `super(`*`paramètres`*`)`.

Exemple 8.2

Les appels à `setX()` et `setY()` ainsi que les initialisations des membres `gr` et `libre` sont communs aux constructeurs de `Proteine` et `Phosphate`. On peut les placer dans un constructeur de leur ancêtre commun `MolMob`. Ce constructeur doit alors être appelé par les constructeurs respectifs des classes dérivées de `MolMob`.

Exercice 8.2

Faire encore « monter d'un cran » l'initialisation de `g`, et les appels à `setX()` et `setY()`, en les positionnant dans le constructeur de `Molecule`.

Attention : la règle générale concernant les constructeurs s'applique ici. En particulier dès que l'on a écrit un constructeur dans la classe ancêtre, il n'y a plus de constructeur par défaut (sans paramètres). Si l'on a besoin d'appeler un constructeur sans paramètres au niveau de cette classe, alors il faut l'écrire.

Exercice 8.3

Apporter les modifications suivantes en choisissant judicieusement le niveau où elles interviennent :

- Doter `Phosphate` et `Proteine` d'un attribut `speed`, qui indique leur vitesse de déplacement, et modifier en conséquence la méthode `bougeAlea()`, qui n'a plus besoin de paramètre mais doit tenir compte de `speed`.
- Créer pour `Phosphate` et `Proteine` de nouveaux constructeurs qui ne prennent pas de coordonnées en paramètres, mais positionnent leur objet aléatoirement.

COMPRENDRE

- La notion d'héritage comme moyen de généraliser les propriétés des objets.
- Les appels de constructeurs dans une hiérarchie de classes.

TD 9 - Héritage II

1. Préliminaires

Avant d'aborder le surclassement, nous allons réintroduire le complexe catalyseur-protéine que nous avons abandonné au TD7. Ceci nécessite tout d'abord de modifier la classe `Complexe` conformément aux autres classes : ajouter un membre de type `Graphics` pour stocker un contexte graphique, et l'inititialiser dans le constructeur (qui prend donc un objet de ce type en argument). De même, un membre de type `Random` est ajouté, et initialisé à la construction.

Nous pouvons maintenant affiner un peu notre modèle. L'idée est qu'au début de la simulation, le complexe n'est pas réellement formé : il contient un catalyseur, mais pas encore de protéine. En pratique, cela se manifeste par les changements suivants dans la classe `Complexe` :

- Un nouveau constructeur permet de créer un complexe sans protéine : il n'initialise pas de `Proteine`, son membre `sub` est égal à `null`. Bien sûr, cet état de choses est susceptible de changer au cours du déroulement du programme.
- Avant d'effectuer un traitement sur le membre `sub`, il faut vérifier qu'il ne vaut pas `null`. Ceci est nécessaire pour les méthodes `transloc()`, `dessine()` et `motInSub()`.
- Enfin, `Complexe` doit maintenant hériter de la classe `Observer`, et donc inclure une méthode `update()`, inspirée de celle que nous avions écrite dans `Catal`. Cette dernière n'est plus un `Observer` et n'a donc plus besoin de méthode `update()`.

Dans `Observation`, le complexe doit être instancié en tenant compte du nouveau constructeur, et (pour l'instant) on ne teste plus la présence du motif de phosphorylation.

2. Transtypage

Le transtypage consiste à transformer un objet d'un type donné en objet d'un autre type. Evidemment, il n'est pas question de transtyper entre types quelconques, mais cette opération peut prendre sons sens dans le contexte d'une hiérarchie de classes.

a) Sur-classement (*upcasting*)

On parle de **sur-classement** lorsqu'un objet est déclaré d'une classe de base donnée, puis instancié dans une classe dérivée de cette classe de base. Comme un objet de la classe dérivée est un objet de la classe de base, le sur-classement est automatique.

Exemple 9.1

Pour manipuler plus d'objets mobiles, nous pouvons créer un tableau de `Proteine` et un tableau de `Phosphate`. Mais tous ces objets sont en fait des instances de `MolMob`. Il est donc possible de déclarer un seul tableau de `MolMob`, et d'en instancier chaque élément avec soit un objet de type `Proteine`, soit un objet de type `Phosphate`.

Si nous utilisons ainsi le sur-classement, `bougeAlea()` ne pose pas de problème : elle est définie dans la classe `MolMob`. Comme toute instance de `Proteine` ou de `Phosphate` est une instance de `MolMob`, la méthode est disponible au niveau des descendants comme de l'ancêtre.

b) Sous-classement (*downcasting*)

Contrairement à `bougeAlea()`, la méthode `vivre()` n'étant pas la même dans `Proteine` et dans `Phosphate`, nous ne l'avons pas écrite dans `MolMob`. Il est alors impossible d'appeler `vivre()` sur un élément du tableau de `MolMob` : le compilateur signale qu'il n'existe pas de méthode `vivre()` définie dans `MolMob`. Pour appeler la méthode correcte pour chaque élément, nous avons le choix entre deux façons de procéder :

- Créer dans `MolMob` une méthode `vivre()`. Le compilateur accepte alors l'appel à `vivre()` sur un élément du tableau de `MolMob`. Mais lors de l'exécution, comme chaque élément de ce tableau est en fait soit une instance de `Proteine`, soit une instance de `Phosphate`, alors ce sont les méthodes de ces classes qui sont appelées. Le contenu de la méthode `vivre()` de `MolMob` n'a donc aucune importance (on écrit une méthode vide).
- « Descendre » dans la hiérarchie vers la classe appropriée. Cette opération, nommée « sous-classement », n'est pas automatique. Il faut effectuer explicitement le transtypage. En pratique, il nous faut tester la classe à laquelle appartient l'objet, à l'aide de l'opérateur **`instanceof`**. Nous pouvons ensuite transtyper l'objet vers cette classe avant d'appeler la méthode `vivre()`.

Nous choisissons la première solution pour l'appel à `vivre()`. La seconde sera utilisée dans l'exercice suivant.

3. Intermède : enrichissement du programme

Notre programme est maintenant prêt à ressembler à une simulation de réaction enymatique. Il n'y manque que l'essentiel : la réaction ! Sa mise en place ne nécessite pas de nouvelle notion, et servira plutôt d'exercice de révision.

Exercice 9.1

Apporter les modifications suivantes :

- Dans un premier temps, nous modifions notre protéine pour représenter sa liaison d'un phosphate. La classe `Proteine` doit être dotée d'un nouvel attribut : `pho`, qui désigne le `Phosphate` auquel elle est liée. Si aucun `Phosphate` n'est lié, alors `pho` prend simplement la valeur `null`. Désormais, l'attribut `config` est inutile : `getConfig()` renvoie '`N`' quand `pho` vaut `null`. Les autres méthodes (y compris les constructeurs) doivent aussi être adaptées à cette nouvelle représentation. En particulier, on peut supprimer `transloc()`.
- Lorsqu'un `Phosphate` est adsorbé, il doit le signaler au complexe catalyseur-protéine. Ce n'est pas difficile, puisque `Phosphate` descend d'`Observable`. Il peut donc appeler la méthode `update()` de `Complexe`, qu'il faut modifier en conséquence : si une protéine de configuration `N` est présente dans le complexe, alors il y a phosphorylation (ce phosphate est alors lié à la protéine), immédiatement suivie de la libération de la protéine phosphorylée. Les déplacements d'un `Phosphate` lié suivent ceux de sa `Proteine`.
  NB : afin de simplifier `update()`, on intégrera dans la méthode `libereProt()` de `Catal` les instructions de positionnement en (x-100,y-100).
- Dans le cas où une protéine phosphorylée est adsorbée sur le catalyseur (un complexe sans substrat), alors le phosphate est libéré, ainsi que la protéine.

4. Classes abstraites

Les classes ancêtres de notre hiérarchie représentent des objets idéaux, que l'on ne peut pas manipuler en tant que tels : que serait une `MolMob` qui ne soit ni un `Phosphate` ni une `Proteine` ? Dans un modèle plus élaboré, il pourrait bien sûr exister d'autres molécules mobiles, mais toutes seraient des dérivées de `MolMob`. En d'autres termes, il serait absurde d'instancier un objet de la classe `MolMob`. Cette particularité se traduit en Java par le fait que nous pouvons faire de `MolMob` une classe abstraite. Il en est de même, à plus forte raison, pour `Molecule`.

Exercice 9.2

Pour rendre abstraites les classes `Molecule` et `MolMob`, il suffit d'écrire **`abstract`** au début de leur déclaration.

Une classe abstraite peut déclarer des méthodes abstraites. Une méthode abstraite est déclarée, mais non spécifiée. Chaque classe dérivée non abstraite doit obligatoirement comporter une spécification de cette méthode. Nous pouvons ainsi déclarer abstraite la méthode `vivre()` dans `MolMob`, ce qui permet d'alléger le code de `Observation`.

Procéder de même pour `dessiner()`.

COMPRENDRE

- le transtypge, vers le haut ou vers le bas.
- l'opérateur `instanceof`.
- les classes et méthodes abstraites.

TD 10 - Échange de données avec un programme

1. Introduction

Jusqu'à présent, tous les programmes que nous avons écrits étaient incapables de communiquer avec l'extérieur. Le seul moyen de transmettre des données (la position d'une molécule, la séquence d'une proteine, etc.) à un objet consiste alors à les inscrire « en dur » dans le programme exécutable qui utilise cet objet. Les valeurs sont transmises en tant que paramètre(s) passés à une méthode : le constructeur (lors de l'instanciation de l'objet), ou une méthode d'accès écrite à cet effet (`setX()`, `setConfig()`, etc.). C'est donc le programmeur et lui seul qui décide des valeurs avec lesquelles fonctionne le programme. Quand on veut modifier ces valeurs, il faut éditer le source, recompiler... tout cela n'est pas très interactif !

2. Ligne de commande

a) Principe

C'est le moyen de communication le plus simple : les paramètres sont passés au programme lors du lancement. La méthode `main()` lit les paramètres sous forme d'un tableau de `String` :

```
public static void main (String args[])
```

Au lancement du programme, le tableau `args[]` est rempli avec les arguments, dans l'ordre où ils sont donnés sur la ligne de commande.

Exemple 10.1

On peut réintroduire la recherche de motif dans le programme de façon à lire le motif à partir de la ligne de commande. On lancera donc le programme avec une commande telle que :

```
java Milieu ILHVGF
```

b) Conversions

Lorsqu'on lance un programme, les paramètres sont des chaînes de caractères, et le type du tableau `args[]` est donc `String`. Si les paramètres sont d'un autre type (des nombres, par exemple), il faut effectuer une **conversion**, pour interpréter la suite de caractères (les chiffres qui représentent un nombre) sous une autre forme (une valeur numérique). On utilise à cet effet les méthodes statiques `parseInt()` de la classe `Integer`, `parseDouble()` de la classe `Double`, etc.

Exemple 10.2

Nous pouvons donner à l'utilisateur le choix du nombre d'itérations de la simulation.

Exercice 10.1

Permettre à l'utilisateur de choisir lui-même le nombre de molécules : instances de `Proteine` et `Phosphate`, en nombre égal.

c) Gestion d'erreurs

Le problème, lorsqu'on effectue une conversion à partir d'une chaîne donnée par l'utilisateur, c'est qu'on ne peut pas être certain par avance que cette chaîne pourra effectivement être convertie : une suite de chiffres est une chaîne de caractère, mais toute chaîne de caractères n'est pas convertible nombre. Par exemple, si on lance le programme précédent avec la commande :

```
java Observation ILHVGF mille 50
```

on obtient un erreur, que Java nomme `NumberFormatException`. Si on veut éviter l'interruption du programme, il faut traiter cette erreur, ou **lever l'exception**. Le mécanisme consiste à indiquer dans le programme ce qui doit être fait si une erreur d'un type donné est rencontrée au cours de l'exécution d'une (ou plusieurs) instruction(s).

Exemple 10.3

Si le nombre d'itérations n'est pas un nombre, mettre la valeur correspondante à 1000.

Exercice 10.2

Si le nombre de molécules n'est pas un nombre, mettre la valeur correspondante à 100.

3. Communication avec les fichiers

Il peut arriver qu'un programme nécessite de longues entrées, qu'il serait fastidieux de taper sur la ligne de commande. De même, les résultats d'un programme peuvent être longs, et nécessiter un stockage. Il s'avère alors utile de lire et écrire des données dans un fichier. Un fichier en Java est un objet de type `File` (contenu dans le package `java.io.*`, qu'il faut importer), qu'on initialise avec une chaîne de caractères qui contient le nom (chemin relatif ou absolu) du fichier :

```
File fich1 = new File("nom_de_fichier");
```

a) Ecriture dans un fichier

Pour écrire dans un fichier, on peut utiliser la classe `FileWriter`. Lorsqu'on crée une instance de `FileWriter`, on l'associe à un fichier :

```
FileWriter fw = new FileWriter(fich1);
```

La classe `FileWriter` propose une méthode « de bas niveau » nommée `write()` qui permet d'écrire un caractère ou une chaîne. On pourrait ainsi écrire :

```
fw.write("Hello !",0,7);  // écrire 7 caractères à partir du 0ème
```

Cette façon de procéder est assez lourde à mettre en œuvre, car il faut explicitement transformer les données à écrire en `String`, en calculer la taille, etc. Heureusement, l'ensemble de ces opérations est réalisé par une classe de plus haut niveau : `PrintWriter`. On dispose alors de méthodes telles que `print()` et `println()`, qui fonctionnent comme nous en avons l'habitude avec `System.out`.

Avec les classes que nous venons de décrire, il est impératif de traiter les erreurs qui pourraient survenir sur les fichiers. Il s'agit des exceptions `FileNotFoundException` et `IOException`.

Enfin, il faut savoir que les fichiers fonctionnent avec un **tampon** afin d'accélérer les accès au disque : lorsqu'une instruction telle que `println(...)` est exécutée, l'écriture n'est pas immédiate. Les données sont en fait stockées dans une zone de la mémoire appelée tampon. Au moment de la fermeture du fichier, ou lorsque le tampon est plein, son contenu est effectivement écrit sur le disque. Il ne faut donc pas oublier de fermer le fichier afin que les données y soient écrites.

Exemple 10.4

Le programme de simulation de réaction enzymatique peut écrire dans un fichier l'évolution au cours du temps du pourcentage de molécules de substrat phosphorylé. Si on désire visualiser les résultats sous forme de courbes, on peut utiliser le programme `gnuplot`. Il faut alors prendre soin de respecter le format d'entrée dans `gnuplot` : on écrit une ligne contenant l'abscisse puis l'ordonnée pour chaque point de la courbe. Pour tracer la courbe, on lance `gnuplot`, puis on lui donne la commande :

```
plot 'nom_du_fichier' with lines
```

Pour quitter `gnuplot` et revenir à l'invite normale, taper `exit`.

b) Lecture depuis un fichier

De même que pour l'écriture, nous disposons d'un tandem de deux classes de haut niveau : un « lecteur de fichier » (`FileReader`) prend les octets dans le fichier, et les stocke dans un tampon jusqu'à ce qu'un un « lecteur de tampon » (`BufferedReader`) les lise (grâce à la méthode `readLine()`) sous forme de chaînes de caractères. On peut ensuite récupérer ces chaînes, les convertir en nombres, etc. Il faut aussi traiter les exceptions `FileNotFoundException` et `IOException`.

Exemple 10.5

Au lieu de n'avoir que des protéines de même séquence, on peut lire la séquence des protéines dans un fichier.

4. Entrées / Sorties à la console

Lorsqu'on passe des paramètres à la ligne de commande ou qu'on lit des données dans un fichier, tout doit être décidé à l'avance, ce qui interdit à l'utilisateur de réagir à des messages du programme. Il peut être préférable de faire lire des valeurs par le programme en cours d'exécution.

Nous avons l'habitude d'utiliser la méthode `println()` de `System.out` pour écrire à l'écran. Il existe de même un **flux** « entrant », nommé sans surprise `System.in`. C'est ce flux qui reçoit directement les caractères tapés au clavier. Ce flux de caractères est en fait analogue à un fichier, et le mécanisme de lecture est équivalent, avec comme seule différence qu'on lit le flux `System.in` avec la classe `InputStreamReader` et qu'on n'a évidemment pas à gérer d'erreur de type `FileNotFoundException`.

Exercice 10.3

Demander à l'utilisateur de choisir le nombre de protéines.

COMPRENDRE

- Les différentes façons de faire communiquer un programme avec son environnement.
- La manipulation des fichiers.
- L'utilisation des exceptions.